Journées Francophones de Programmation par contraintes

2009

Orléans

| 3 - 5 juin 09 |

Cinquièmes Journées Francophones de Programmation par Contraintes

Orléans, 3-5 juin 2009

Éditeur : Yves Deville

UNIVERSITE D'ORLEANS

LIFO

Actes des Cinquièmes Journées Francophones de Programmation par Contraintes

3–5 juin 2009 – Orléans

Organisation

Université d'Orléans, Laboratoire d'Informatique Fondamentale d'Orléans

Président des journées

Alexandre Tessier, Université d'Orléans

Président du comité de programme

Yves Deville, UCLouvain, Belgique

Comité de programme

Philippe Baptiste,	LIX Palaiseau
Mustapha Belaissaoui,	Univ. Hassan I Maroc
Nicolas Beldiceanu,	EMN/LINA Nantes
Christian Bessiere,	CNRS/LIRMM Univ. Montpellier
Hadrien Cambazard,	4C Cork,Ireland
Philippe Chatalic,	LRI Orsay
Thi Bich Hanh Dao,	LIFO Orleans
Simon de Givry,	INRA Toulouse
Sophie Demassey,	EMN/LINA Nantes
François Fages,	INRIA Rocquencourt
Pierre Flener,	IT Uppsala Univ., Suède
Alexandre Godsztejn,	CNRS/LINA Nantes
Arnaud Gotlieb,	IRISA Rennes
Youssef Hamadi,	Microsoft Research Cambridge Royaume Uni
Mhand Hifi,	MIS Amiens
Daniel Le Berre,	CRIL Université d'Artois
Christophe Lecoutre,	CRIL Université d'Artois
Laurent Michel,	Univ. Connect. USA

Wadi Naanaa,	Univ. Monastir Tunisie
Lionel Paris,	LSIS Marseille
Laurent Perron,	Google Paris
Cédric Pralet,	Onera Toulouse
Philippe Refalo,	ILOG Sophia-Antipolis
Gilles Pesant,	Polytechnique Montréal
Michel Rueher,	I3S/CNRS Université de Nice Sophia Antipolis
Frédéric Saubion,	LERIA Angers
Pierre Schaus,	UCLouvain Belgique
Christine Solnon,	LIRIS Lyon
Cyril Terrioux,	LSIS Marseille
Alexandre Tessier,	LIFO Orléans
Gilles Trombettoni,	INRIA Sophia
Charlotte Truchet,	LINA Université de Nantes
Michel Vasquez,	EMA Nîmes
Stéphane Zampelli,	EMN/LINA Nantes
Bruno Zanuttini,	GREYC Univ. de Caen Basse-Normandie

Relecteurs additionnels

Ignacio Araya, Lucas Bordeaux, Eric Bourreau, Florence Charreteur, Raphaël Chenouard, Marc Christie, Remi Coletta, Mickael Delahaye, Khalil Djelloul, Renaud Dumeur, Djamal Habet, Said Jabbour, Philippe Jégou, Frédéric Lardeux, Nadjib Lazaar, Matthieu Lopez, Toni Mancini, Julien Martin, Jean-Noël Monette, Samba Ndojh Ndiaye, Bertrand Neveu, Richard Ostrowski, Alexandre Papadopoulos, Justin Pearson, Cédric Piette, Philippe Refalo, Jérémie Vautard, Guillaume Verger, Petr Vilim, Alessandro Zanarini

Comité d'organisation

Thi-Bich-Hanh Diep-Dao
Denys Duchier
AbdelAli Ed-Dbali
Willy Lesaint
Matthieu Lopez
Florence Maubert
Isabelle Renard
Alexandre Tessier
Jérémie Vautard

Exposés invités

Laurent Trilling,	TIMC-IMAG, Université Joseph Fourier, Grenoble
Pascal Van Hentenryck,	Brown University, Providence, USA

Préface

La programmation par contraintes (PC) est porteuse d'une grande ambition : celle d'offrir des langages de modélisation permettant de décrire et de résoudre efficacement des problèmes combinatoires complexes tels que la planification, l'allocation de ressources, le routage de véhicules ou l'ordonnancement. La PC propose une nouvelle approche pour l'optimisation combinatoire ; orthogonale et complémentaire à la recherche opérationnelle traditionnelle. Une des caractéristiques essentielles de la PC est la séparation entre l'expression d'un modèle, sous la forme d'un ensemble de contraintes, et la spécification d'une stratégie d'exploration de l'espace des solutions. Cette séparation offre le double avantage de réduire le temps de développement et d'offrir une approche modulaire : un modèle existant peut être affiné par l'ajout de contraintes supplémentaires. La PC permet de considérer des domaines finis et continus ; elle offre des méthodes de recherche complète ainsi que des méthodes de recherche locale ; elle permet de combiner différentes approches ainsi que d'intégrer des méthodes de programmation mathématiques.

Le succès de la PC dans le monde industriel s'est affirmé dans des domaines aussi divers que le transport, les télécommunications, le sport, les services, la production, la gestion du personnel et la conception. Plusieurs outils commerciaux sont aujourd'hui disponibles et intègrent les technologies les plus récentes. La PC peut aujourd'hui s'ouvrir vers de nouvelles applications innovantes telles que l'optimisation stochastique et les problèmes de décision avec incertitude.

Les Journées Francophones de Programmation par Contraintes (JFPC) se veulent un lieu convivial de rencontres, discussions et échanges scientifiques, pour la communauté francophone, en particulier entre thésards, chercheurs plus confirmés et industriels. Les JFPC sont patronnées par l'Association Française pour la Programmation par Contraintes (AFPC) qui réunit les chercheurs, ingénieurs et enseignants du domaine, souvent issus de l'informatique théorique, de la programmation logique, de l'intelligence artificielle, de l'algorithmique ou de la recherche opérationnelle. Cette cinquième édition des JFPC a lieu à Orléans du 3 au 5 juin 2009. Elle fait suite aux manifestations antérieures qui se sont tenues à Nantes (2008), Rocquencourt (2007), Nîmes (2006) et Lens (2005). Les JFPC sont issues de la fusion des conférences JFPLC (Journées Francophones de la Programmation Logique avec Contraintes) nées en 1992 et JNPC (Journées Nationales sur la Résolution Pratique de Problèmes NP-Complets) nées en 1994. Avant cette fusion, le rapprochement entre les deux communautés s'est traduit, depuis 1998, par l'organisation conjointe des JFPLC et des JNPC : Angers (2004), Amiens (2003), Nice (2002), Toulouse et Paris (2001), Marseille (2000), Lyon (1999) et Nantes (1998).

Quarante-sept articles ont été soumis à l'édition 2009 des JFPC, provenant de France, de Belgique, d'Angleterre, des Etats-Unis, de Tunisie, du Chili, de Chine, d'Algérie, du Brésil, du Danemark et du Maroc. Le comité de programme en a sélectionné trente-neuf, auxquels se sont ajoutées deux conférences invitées, par Laurent Trilling, IMC-IMAG, Université Joseph Fourier, Grenoble, et Pascal Van Hentenryck, Brown University, Providence, USA. Le programme compte également une demi-journée industrielle, composée de présentations de produits et de tutoriels.

Je tiens à remercier les auteurs des articles soumis ainsi que les membres du comité de programme et tous les relecteurs, pour le soin qu'ils ont apporté à l'évaluation de ces articles, leurs critiques constructives et les discussions sur EasyChair. Merci à Gilles Trombettoni, ancien Président du comité de programme pour ses précieux conseils. Je remercie également tous les membres du comité d'organisation, et tout d'abord son Président, Alexandre Tessier, pour son efficacité dans l'organisation de ces journées. Un merci tout particulier à Matthieu Lopez qui a beaucoup aidé pour l'organisation et en particulier dans son rôle de webmaster. Merci également à Isabelle Renard et Florence Maubert qui ont aidé pour les aspects administratifs, Willy Lesaint pour l'organisation de la demi-journée industrielle, Catherine Tallon webdesigner à l'Université d'Orléans, Thi-Bich-Hanh Diep-Dao, Denys Duchier, AbdelAli Ed-Dbali et Jérémie Vautard, membres du comité d'organisation.

Enfin nous remercions nos partenaires institutionnels et industriels qui nous ont soutenu financièrement : l'INRIA, le LIFO, Ilog, la Région Centre, l'Université d'Orléans, Bouygues, la ville d'Orléans, GDR GPL du CNRS, Cosytec, le Département du Loiret, KLS OPTIM, Dynadec et l'AFPC.

Yves Deville Président du comité de programme des JFPC'09

Table des matières

Exposés invités **1**

Approche déclarative de la modélisation de réseaux de régulation de gènes discrets : expériences et perspectives
Laurent Trilling 1

Online Stochastic Combinatorial Optimization : Progress and Opportunities
Pascal Van Hentenryck 3

Recherche (1) **5**

Une généralisation de l'approche Cyclic-Clustering pour la résolution de CSP structurés
Cédric Pinto, Cyril Terrioux 5

Algorithme des Poupées Russes exploitant une décomposition arborescente
Marti Sanchez, David Allouche, Simon de Givry, Thomas Schiex 15

Comptage de solutions en exploitant la structure du graphe de contraintes
Aurélie Favier, Simon de Givry, Philippe Jégou 25

Stratégies hybrides pour des décompositions optimales et efficaces
Philippe Jégou, Samba Ndojh Ndiaye, Cyril Terrioux 35

Modélisation et langages **45**

Aeon : Synthèse d'algorithmes d'ordonnancement à partir de modèles de haut niveau
Jean-Noël Monette, Yves Deville, Pascal Van Hentenryck 45

Reformulation de problèmes de satisfaction de contraintes sur métamodèles
Raphaël Chenouard, Ricardo Soto, Laurent Granvilliers 55

Vers une théorie du test des programmes à contraintes
Lazaar Nadjib, Gotlieb Arnaud, Lebbah Yahia 65

CHR modulaire avec ask et tell
François Fages, Thierry Martinez, Cleyton Rodrigues 75

SAT (1) **85**

Une approche basée sur la décomposition arborescente pour la résolution d'instances SAT structurées
Djamal Habet, Lionel Paris, Cyril Terrioux 85

Localiser des sources d'incohérence spécifiques sans les calculer toutes
Eric Gregoire, Bertrand Mazure, Cédric Piette 95

Subsumption dirigée par l'analyse de coflits
Youssef Hamadi, Said Jabbour, Lakhdar Sais 105

Contraintes globales et qualitatives **115**

Des contraintes globales prêtes à brancher
Guillaume Richaud, Lorca Xavier, Narendra Jussien 115

Séquencer et compter avec la contrainte multicost-regular
Julien Menana, Sophie Demassey 125

Fusion de réseaux de contraintes qualitatives par morceaux
Jean-François Condotta, Souhila Kaci, Pierre Marquis, Nicolas Schwind 135

Applications **145**

Génération rapide de scénarios géophysiques par satisfaction de contraintes pour la localisation des séismes
Jean-Philippe Poli, Anthony Larue, David Mercier, Carole Maillard, Jocelyn Guilbert 145

Relaxation de contraintes globales pour la modélisation de problème d'allocation d'infirmières
Jean-Philippe Métivier, Patrice Boizumault, Samir Loudni 155

Problème d'équilibre des charges de travail dans l'affectation de patients aux infirmières
Pierre Schaus, Pascal Van Hentenryck, Jean-Charles Régin 165

Combiner contraintes et modèles pour le traitement de langage contrôlé
Mathias Kleiner, Patrick Albert, Jean Bezivin 175

Recherche locale **185**

Génération et contrôle autonomes d'opérateurs pour les algorithmes évolutionnaires
Jorge Maturana, Frédéric Lardeux, Frédéric Saubion 185

Un modèle réactif pour l'optimisation par colonies de fourmis : application à la satisfaction de contraintes
Madjid Khichane, Christine Solnon, Patrick Albert 195

Une approche de contrôle autonome pour la recherche locale
Julien Robet, Frédéric Lardeux, Frédéric Saubion 205

Analyse de conflits dans le cadre de la recherche locale
Audemard Gilles, Lagniez Jean-Marie, Mazure Bertrand, Sais Lakhdar 215

SAT (2) **225**

RB-SAT : Un nouveau modèle SAT basé sur les codages du modèle RB
Haibo Huang, Chu Min Li, Nouredine Ould Mohamedou, Ke Xu 225

Détection de fonctions booléennes pour la preuve d'inconsistance
Richard Ostrowski, Lionel Paris 237

Pourquoi les solveurs SAT modernes se piquent-ils contre des cactus ?
Gilles Audemard, Mouny Samy Modeliar, Laurent Simon 245

Contraintes quantifiées **255**

Décidabilité de contraintes du premier ordre par contraintes duales
Khalil Djelloul 255

Problèmes d'optimisation avec des contraintes quantifiées
Marco Benedetti, Arnaud Lallouet, Jérémie Vautard 265

Sémantique et calcul syntaxique pour les formules booléennes quantifiées
Igor Stéphan 275

Apprentissage et CSP dynamiques **285**

Un calcul de Viterbi pour un Modèle de Markov Caché Contraint
Matthieu Petit, Henning Christiansen 285

Réordonnancement dynamique basé sur l'apprentissage
Youssef Hamadi, Said Jabbour, Lakhdar Sais 295

CSP dynamiques pour la génération de tests de systèmes réactifs
Christophe Junke, Benjamin Blanc 305

Consistence et décision **315**

Un schéma générique pour intégrer des consistances fortes dans les solveurs de contraintes
Julien Vion, Thierry Petit, Narendra Jussien 315

Consistance Duale et réseaux non-binaires
Julien Vion 325

Utilisation de sous-systèmes carrés et denses pour le filtrage de CSP numériques
Ignacio Araya, Gilles Trombettoni, Bertrand Neveu 335

Un algorithme de décision dans l'algèbre des arbres finis ou infinis et des queues
Thi-Bich-Hanh Dao 345

Recherche (2) **355**

Cohérences basées sur les valeurs en échec
Christophe Lecoutre, Olivier Roussel 355

Identification et exploitation d'états partiels inconsistants
Christophe Lecoutre, Sebastien Tabary, Vincent Vidal 365

Algorithme de décomposition de domaine pour la satisfaction et l'optimisation de contraintes
Wady Naanaa, Maher Helaoui, Bechir Ayeb 375

Recherche de la substituabilité par l'arc-cohérence de singleton
Dominique D'Almeida, Lakhdar Sais 385

Index

Albert Patrick, 175, 195
Allouche David, 15
Araya Ignacio, 335
Audemard Gilles, 215, 245
Ayeb Bechir, 375

Benedetti Marco, 265
Bezivin Jean, 175
Blanc Benjamin, 305
Boizumault Patrice, 155

Chenouard Raphaël, 55
Christiansen Henning, 285
Condotta Jean-François, 135

D'Almeida Dominique, 385
Dao Thi-Bich-Hanh, 345
de Givry Simon, 15, 25
Demassey Sophie, 125
Deville Yves, 45
Djelloul Khalil, 255

Fages François, 75
Favier Aurélie, 25

Gotlieb Arnaud, 65
Granvilliers Laurent, 55
Gregoire Eric, 95
Guilbert Jocelyn, 145

Habet Djamal, 85
Hamadi Youssef, 105, 295
Helaoui Maher, 375
Huang Haibo, 225

Jégou Philippe, 25, 35
Jabbour Said, 105, 295
Junke Christophe, 305
Jussien Narendra, 115, 315

Kaci Souhila, 135
Khichane Madjid, 195
Kleiner Mathias, 175

Lagniez Jean-Marie, 215
Lallouet Arnaud, 265
Lardeux Frédéric, 185, 205
Larue Anthony, 145
Lazaar Nadjib, 65
Lebbah Yahia, 65
Lecoutre Christophe, 355, 365
Li Chu Min, 225
Lorca Xavier, 115

Loudni Samir, 155

Métivier Jean-Philippe, 155
Maillard Carole, 145
Marquis Pierre, 135
Martinez Thierry, 75
Maturana Jorge, 185
Mazure Bertrand, 95, 215
Menana Julien, 125
Mercier David, 145
Modeliar Mouny Samy, 245
Mohamedou Nouredine Ould, 225
Monette Jean-Noël, 45

Naanaa Wady, 375
Ndiaye Samba Ndojh, 35
Neveu Bertrand, 335

Ostrowski Richard, 237

Paris Lionel, 85, 237
Petit Matthieu, 285
Petit Thierry, 315
Piette Cédric, 95
Pinto Cédric, 5
Poli Jean-Philippe, 145

Régin Jean-Charles, 165
Richaud Guillaume, 115
Robet Julien, 205
Rodrigues Cleyton, 75
Roussel Olivier, 355

Sais Lakhdar, 105, 215, 295, 385
Sanchez Marti, 15
Saubion Frédéric, 185, 205
Schaus Pierre, 165
Schiex Thomas, 15
Schwind Nicolas, 135
Simon Laurent, 245
Solnon Christine, 195
Soto Ricardo, 55
Stéphan Igor, 275

Tabary Sebastien, 365
Terrioux Cyril, 5, 35, 85
Trilling Laurent, 1
Trombettoni Gilles, 335

Van Hentenryck Pascal, 3, 45, 165
Vautard Jérémie, 265
Vidal Vincent, 365
Vion Julien, 315, 325

Xu Ke, 225

Approche déclarative de la modélisation de réseaux de régulation de gènes discrets : expériences et perspectives

Laurent Trilling

Laboratoire TIMC-IMAG (Techniques de l'Ingénierie Médicale et de la Complexité)
Université Joseph Fourier (Grenoble I)
Domaine de la Merci, 38710 La Tronche, France
laurent.trilling@imag.fr

Ce type d'activité se situe dans une perspective bioinformatique post-génomique pour répondre au défi représenté par la compréhension des réseaux biologiques. Il s'agit d'abord, à partir de connaissances, d'observations biologiques ou même d'hypothèses sur un phénomène d'aboutir à des modèles cohérents de réseaux, c'est-à-dire des systèmes complexes en biologie. Dans un second temps, l'objectif est celui de toute modélisation : émettre, à partir de ces modèles cohérents, des prédictions bien fondées, essentielles désormais aux biologistes pour poursuivre leur analyse par des expérimentations.

Traditionnellement, on utilise des équations différentielles pour modéliser ce type de réseaux. Mais il apparaît qu'en l'absence de données complètes et précises, des approches discrètes peuvent apporter des informations très intéressantes quant aux comportements dynamiques tels que les états stationnaires et les cycles. Bien sur, la discrétisation doit représenter une abstraction pertinente. C'est le cas pour le formalisme introduit par René Thomas, auquel nous nous intéressons, pour décrire les réseaux de régulation de gènes.

D'un point de vue analyse, une vision " programmation par contraintes " présente l'intérêt de sortir de la méthode d'investigation usuelle, assez "artisanale" : celle bâtie sur la construction d'un premier modèle instancié qu'on confronte avec les observations/hypothèses pour affiner par essai/erreur le modèle original. A l'opposé, l'idée consiste à spécifier formellement la structure du réseau et les contraintes décrivant les observations/hypothèses. De cette façon, on peut adresser des requêtes allant de la simulation à l'inférence des paramètres définissant le réseau en passant par des intermédiaires où les paramètres et les comportements sont partiellement connus. Ce n'est plus un seul modèle qui est considéré, mais une classe.

Nous présenterons essentiellement une méthode d'analyse rendue possible par cette approche déclarative. Elle est basée sur quatre étapes : construction d'une spécification initiale comportant un maximum d'hypothèses biologiques, vérification de cohérence et correction éventuelle par relâchement de contraintes, production de propriétés appartenant des langages préalablement définis, addition (resp. retrait) de contraintes suivant les résultats et retour à la seconde (resp. troisième) étape. Cette méthode sera illustrée par son application au ré-examen fructueux d'une modélisation du stress carboné chez la bactérie *E.coli*. On exposera à cette occasion comment exprimer sous forme de contraintes des caractéristiques biologiques telles que les compositions d'interactions entre gènes et les comportements dynamiques. Les problèmes de mise en oeuvre et de performance seront aussi abordés via cette expérience.

Pour terminer, nous évoquerons les nombreux développements envisagés sur le plan des formalismes, des méthodes algorithmiques et des applications.

Ces travaux ont été réalisés en collaboration avec Delphine Ropers (INRIA-Rhones-Alpe), Sébastien Corblin, Eric Fanchon et Sébastien Tripodi (TIMC-IMAG).

Online Stochastic Combinatorial Optimization: Progress and Opportunities

Pascal Van Hentenryck

Brown University, Box 1910, Providence, RI 02912, USA
pvh@cs.brown.edu

Résumé

Progress in telecommunications offers tremendous challenges and opportunities for optimization technology. The challenges is to take operational decisions under uncertainty, while the opportunities may fundamentally affect the future role of optimization. This talk reviews progress in this area, survey challenging applications, and study the fundamental role that constraint programming can play in addressing the needs.

Une généralisation de l'approche Cyclic-Clustering pour la résolution de CSP structurés

Cédric Pinto **Cyril Terrioux**

LSIS - UMR CNRS 6168
Université Paul Cézanne (Aix-Marseille 3)
Avenue Escadrille Normandie-Niemen
13397 Marseille Cedex 20 (France)
{cedric.pinto, cyril.terrioux}@lsis.org

Résumé

Nous proposons une nouvelle méthode pour résoudre des CSP structurés, qui généralise et améliore l'approche Cyclic-Clustering [6]. D'abord, l'ensemble coupe-cycle et la décomposition arborescente du réseau de contraintes, qui sont employés pour tirer profit de la structure du CSP, sont calculés indépendamment de la notion quelque peu contraignante de sous-graphe triangulé induit. Ensuite, contrairement à Cyclic-Clustering, la méthode proposée permet de résoudre le CSP induit par la décomposition arborescente sans avoir nécessairement instancié au préalable toutes les variables de la partie coupe-cycle. La résolution d'un tel CSP peut être effectuée grâce à la méthode BTD [8] comme dans [9]. Dans la mesure où BTD mémorise et exploite des (no)goods structurels, nous nous intéressons ensuite aux conditions qui rendent possible la réutilisation de (no)goods structurels enregistrés lors d'appels précédents à BTD. Ces conditions sont alors exploitées dans une version dédiée de BTD de sorte à profiter autant que possible des informations précédemment mémorisées.

Du point de vue théorique, la méthode proposée permet de garantir une borne de complexité en temps relative à des paramètres liés à l'ensemble coupe-cycle et à la décomposition arborescente employés. Concernant l'intérêt pratique, nous pouvons espérer détecter plus tôt les échecs et éviter certaines redondances dans la recherche. Cet intérêt pratique est évalué lors d'expériences préliminaires.

Abstract

We propose a new method for solving structured CSPs which generalizes and improves the Cyclic-Clustering approach [6]. First, the cutset and the tree-decomposition of the constraint network, which are used for taking advantage of the CSP structure, are computed independently of the notion of triangulated induced subgraph. Then, unlike Cyclic-Clustering, our method can try to solve the tree-decomposition part of the problem without having assigned all the variables of the cutset. Regarding the solving of the tree-decomposition part, we use the BTD method [8] like in [9]. As BTD records and exploits structural (no)goods, we provide some conditions which make possible the use of structural (no)goods recorded during previous calls of BTD and we implement them in a dedicated version of BTD. By so doing, from a theoretical viewpoint, we can provide a theoretical time complexity bound related to parameters of the cutset and the tree-decomposition and, from a practical viewpoint we expect to detect failures earlier and to avoid more redundancies in the search. This practical interest is assessed in some preliminary experiments.

1 Préliminaires

Le formalisme des problèmes de satisfaction de contraintes (CSP) offre un cadre de formalisation extrêmement puissant pour l'expression et la résolution d'une multitude de problèmes, en particulier de nombreux problèmes académiques ou issus du monde réel (par exemple les problèmes de coloration de graphes, d'ordonnancement, d'affectation de fréquence, ...). Un *problème de satisfaction de contraintes* (X, D, C, R) est défini comme un ensemble de variables $X = \{x_1, \ldots x_n\}$ soumis à un ensemble de contraintes $C = \{c_1, \ldots, c_m\}$. A chaque variable x_i est

associé un domaine d_i issu de l'ensemble de domaines $D = \{d_1, \dots d_n\}$. Le domaine fini d_i contient chacune des valeurs possibles pour x_i. Une contrainte $c_i \in C$ est un sous-ensemble ordonné de variables, $c_i = (x_{i_1}, \dots x_{i_{a_i}})$ (le nombre a_i de variables impliquées dans la contrainte c_i est appelé l'arité de cette contrainte). Notons que nous utilisons la même notation pour désigner la contrainte c_i et l'ensemble des variables sur lesquelles elle porte. A chaque contrainte c_i est associée une relation $r_{c_i} \in R$ définissant les combinaisons de valeurs autorisées pour les variables soumises à la contrainte c_i ($r_{c_i} \subseteq d_{i_1} \times \dots \times d_{i_{a_i}}$). Soit $Y = \{x_1, \dots x_k\}$ un sous-ensemble de X. Une *affectation* $\mathcal{A}$ sur Y est un tuple $(v_1, \dots, v_k)$ de $d_1 \times \dots \times d_k$. Nous écrirons également $\mathcal{A}$ sous la forme plus explicite $\{x_1 \leftarrow v_1, ..., x_i \leftarrow v_k\}$. Ensuite, nous noterons $\mathcal{A}_1 \subseteq \mathcal{A}_2$ si l'affectation A_2 est une extension de l'affectation A_1 (i.e. nous avons $\mathcal{A}_1 = \{x_1 \leftarrow v_1, ..., x_i \leftarrow v_i\}$ et $\mathcal{A}_2 = \{x_1 \leftarrow v_1, ..., x_i \leftarrow v_i, ..., x_{i+j} \leftarrow v_{i+j}\}$ avec $j \geq 0$). Une affectation $\mathcal{A}$ sur Y satisfait une contrainte $c \in C$ telle que $c \subseteq Y$ si $\mathcal{A}[c] \in r_c$ où $\mathcal{A}[c]$ désigne la restriction de $\mathcal{A}$ aux variables sur lesquelles porte c. $\mathcal{A}$ est dite *consistante* si elle satisfait chaque contrainte c telle que $c \subseteq Y$. Une solution est une affectation de chacune des variables qui satisfait toutes les contraintes. Déterminer s'il existe une solution constitue un problème NP-Complet. Nous noterons $Sol(\mathcal{P})$ l'ensemble des solutions du CSP $\mathcal{P}$. Pour des raisons de simplicité, nous ne considérerons, dans la suite de cet article, que le cas des CSP binaires (c'est-à-dire des CSP dont chaque contrainte porte sur deux variables exactement). Néanmoins, ces travaux peuvent s'étendre aux CSP n-aires.

Les méthodes utilisées habituellement pour résoudre des CSP (comme, par exemple, les méthodes Forward Checking [5] et MAC [11]) reposent généralement sur une recherche énumérative de type backtracking. Cette approche, souvent efficace en pratique, a une complexité en temps exponentielle en $O(m.d^n)$ (noté $O(exp(n))$) pour une instance ayant n variables et m contraintes et dont le plus grand domaine possède d valeurs. Plusieurs travaux ont été développés afin d'améliorer cette complexité théorique en exploitant des caractéristiques particulières de l'instance. Généralement, ils reposent sur certaines propriétés structurelles du CSP. La structure d'un CSP (X, D, C, R) peut être représentée par le graphe (X, C), appelé le *graphe de contraintes*. Dans ce contexte, la notion de décomposition arborescente [10] joue un rôle clé. Une *décomposition arborescente* d'un graphe $G = (X, C)$ est un couple(E, T) où $T = (I, F)$ est un arbre dont I est l'ensemble des nœuds et F celui des arêtes et $E = \{E_i : i \in I\}$ une famille de sous-ensembles de X telle que chaque sous-ensemble (appelé cluster) E_i est un nœud de T et vérifie : (i) $\cup_{i \in I} E_i = X$, (ii) pour chaque arête $\{x, y\} \in C$, il existe $i \in I$ avec $\{x, y\} \subseteq E_i$, et (iii) pour tout $i, j, k \in I$, si k est sur une chaîne entre i et j dans T, alors $E_i \cap E_j \subseteq E_k$. La largeur w d'une décomposition arborescente (E, T) est égale à $max_{i \in I}|E_i| - 1$. La *largeur d'arbre* (ou *tree-width*) w^* de G est la largeur minimale sur toutes les décompositions arborescentes de G. D'une part, cette notion conduit à une des meilleures bornes connues de complexité théorique en temps, à savoir $O(exp(w^* + 1))$ avec w^* la largeur d'arbre. Plusieurs méthodes (par exemple [4, 8]) ont été proposées pour atteindre cette borne. Elles visent à regrouper les variables entre elles (en formant des clusters) de sorte que l'arrangement en clusters forme un arbre. D'autre part, les décompositions arborescentes sont également employées dans d'autres méthodes structurelles, comme la méthode Cyclic-Clustering [6]. Plus précisément, la méthode Cyclic-Clustering consiste à instancier un sous-ensemble de variables (appelé ensemble coupe-cycle) tel que le graphe de contraintes du problème réduit aux variables non instanciées est un arbre de cliques (qui correspond donc à une décomposition arborescente).

D'un point de vue théorique, atteindre la meilleure borne de complexité théorique en temps requiert de calculer une décomposition arborescente optimale (c'est-à-dire une décomposition arborescente ayant une largeur minimale), ce qui constitue un problème NP-Difficile [1]. En pratique, il semble difficilement concevable de résoudre un problème NP-Difficile comme étape préliminaire de la résolution d'un problème NP-Complet. Aussi, au lieu d'utiliser une méthode exacte, on fait plutôt appel, en général, à des méthodes heuristiques. Souvent, ces méthodes fournissent de bonnes (voire de très bonnes) approximations d'une décomposition arborescente optimale quand le graphe de contraintes possède une petite largeur d'arbre. Des méthodes comme BTD [8] sont alors tout indiquées pour résoudre de tels problèmes. En revanche, quand la structure du graphe de contraintes est moins "jolie" (et donc que la largeur d'arbre est plus importante), ces méthodes heuristiques produisent trop souvent des décompositions arborescentes avec une largeur importante et conduisent ainsi à des approximations très grossières d'une décomposition arborescente optimale. Dans un tel cas, exploiter une méthode de résolution comme Cyclic-Clustering peut se révéler plus intéressant et surtout plus adapté. Cyclic-Clustering repose sur un sous-ensemble V de sommets (et donc de variables), appelé coupe-cycle, du graphe (X, C), tel que le sous-graphe $(X - V, \{\{x, y\} \in C \; tel \; que \; x, y \in X - V\})$ induit par $X - V$ soit triangulé (c'est-à-dire qu'il ne possède pas de cycle de longueur supérieure ou égale à 4 sans une arête joignant deux sommets non consécutifs dans le cycle). La partie triangulée du graphe de contraintes correspond en fait à une décomposition arborescente. A titre d'exemple, la figure 1(a) présente un graphe ayant 19 sommets. L'ensemble $\{y_1, y_2\}$ forme un coupe-cycle de ce graphe puisque le sous-graphe induit par $\{x_1, \dots, x_{17}\}$ est triangulé. Dans [9], deux implémentations de Cyclic-Clustering, appelées respectivement CC-

BTD_1 et CC-BTD_2, ont été proposées. Toutes deux résolvent d'abord la partie du problème associée à l'ensemble coupe-cycle avec un algorithme énumératif classique (comme FC par exemple), puis la partie associée au sous-graphe triangulé induit en employant la méthode BTD. CC-BTD_2 se distingue de CC-BTD_1 par un appel préliminaire à BTD avant d'entamer la résolution de la partie coupe-cycle. En procédant ainsi, les nogoods mémorisés lors de cet appel préliminaire à BTD restent valides durant l'ensemble de la résolution et donc dans les différents appels ultérieurs à BTD. L'approche Cyclic-Clustering (et plus particulièrement ses deux implémentations) possède plusieurs limites. D'une part, les informations mémorisées, sous forme de goods et de nogoods structurels durant les différents appels à BTD, ne sont pas exploitées ultérieurement (hormis les nogoods enregistrés durant l'appel préliminaire à BTD pour CC-BTD_2), conduisant ainsi à visiter plusieurs fois le même espace de recherche. D'autre part, la partie triangulée doit être calculée en exploitant la notion de sous-graphe triangulé induit (TIS).

Dans cet article, nous proposons une généralisation de CC-BTD, appelée CC-BTD-gen. A l'image de l'approche Cyclic-Clustering, CC-BTD-gen se base sur un ensemble coupe-cycle et une décomposition arborescente du graphe de contraintes. Toutefois, la décomposition arborescente utilisée peut être calculée avec n'importe quelle méthode, que cette méthode exploite ou non la notion de TIS. S'affranchir de la notion de TIS permet, par exemple, de pouvoir calculer un coupe-cycle puis d'en déduire une décomposition arborescente pour les variables qui ne participent pas au coupe-cycle. Concernant la résolution, CC-BTD-gen exploite une version dédiée de BTD qui permet de tirer profit d'une partie des (no)goods mémorisés lors d'appels antérieurs à BTD, permettant ainsi d'éviter, en pratique, un certain nombre de redondances dans la recherche. Enfin, nous avons constaté que CC-BTD instancie de façon consistante toutes les variables de la partie coupe-cycle avant de résoudre la partie associée à la décomposition arborescente, même si après avoir instancié quelques variables du coupe-cycle, il n'est plus possible de trouver une solution sur la partie triangulée. Par conséquent, afin d'éviter un tel inconvénient, CC-BTD-gen a la possibilité d'appeler BTD après avoir instancié de façon consistante seulement une partie des variables du coupe-cycle. Ainsi, si le sous-problème associé à la décomposition arborescente possède une solution, la recherche peut continuer sur les variables non instanciées du coupe-cycle. Sinon, on peut remettre en cause immédiatement l'affectation courante sur le coupe-cycle. Dans les deux cas, des goods et des nogoods pourront être enregistrés et réutilisés plus tard.

Ce papier est organisé ainsi. La section 2 présente le cadre formel de CC-BTD-gen tandis que la section 3 décrit l'algorithme lui-même. Ensuite, nous évaluons expérimentalement l'intérêt de cette approche dans la section 4. Enfin, nous concluons et discutons des travaux connexes ou à venir dans la section 5.

2 Cadre formel

Dans cette section, nous allons décrire le cadre formel requis pour pouvoir présenter proprement CC-BTD-gen. Ce cadre est ici défini dans un contexte général avant d'être utilisé ensuite dans le contexte spécifique de CC-BTD-gen où Y désignera l'ensemble coupe-cycle et $X - Y$ l'ensemble des variables impliquées dans la décomposition arborescente. Dans la suite, nous considérons un CSP $\mathcal{P} = (X, D, C, R)$. Dans un premier temps, nous définissons la notion de sous-problème induit par un sous-ensemble Y de variables.

Définition 1 *Soit $Y \subseteq X$ un sous-ensemble de variables. Le **CSP induit par** Y est le CSP (Y, D_Y, C_Y, R_Y) où $D_Y = \{d_i \in D | x_i \in Y\}$, $C_Y = \{c_{ij} = \{x_i, x_j\} \in C | x_i, x_j \in Y\}$ et $R_Y = \{r_{ij} \in R | c_{ij} \in C_Y\}$.*

Dans les définitions et propriétés suivantes, nous considérerons les notations suivantes :

- Y_1, Y_2, Y et Z seront des sous-ensembles de X tels que $Y_1 \subseteq Y$, $Y_2 \subseteq Y$, $Y \subseteq X$ et $Z \subseteq X - Y$,
- $\mathcal{A}_1$ une affectation sur Y_1, A_2 sur Y_2 et $\mathcal{A}$ sur Y,
- TD la décomposition arborescente utilisée pour le CSP $\mathcal{P}(X - Y)$,
- $S_j = E_i \cap E_j$ un séparateur de TD avec E_i et E_j deux clusters de TD tels que E_j soit un fils de E_i,
- $Desc(E_j)$ l'ensemble des variables appartenant à la descendance du cluster E_i enracinée en E_j.

Nous proposons maintenant une définition certes limitée, mais suffisante, de l'effet du filtrage effectué par l'algorithme Forward-Checking (FC [5]).

Définition 2 *Le filtrage résultant de l'affectation $\mathcal{A}$ accompli par FC est l'opération qui consiste à supprimer du domaine d_i de chaque variable non instanciée x_i les valeurs qui deviennent incompatibles vis-à-vis d'au moins une contrainte $\{x_i, y\}$ avec y une variable instanciée dans $\mathcal{A}$. Plus formellement, $d_i^{\mathcal{A}} = \{v \in d_i | \forall c = \{x_i, y\} \in C,\ (v, w) \in r_c\ avec\ w\ la\ valeur\ affectée\ à\ y\ dans\ \mathcal{A}\}$.*

En d'autres mots, $d_i^{\mathcal{A}}$ désigne le domaine courant d'une variable non instanciée x_i obtenu par l'application du filtrage de FC après chaque affectation d'une variable de $\mathcal{A}$. Avec le même objectif de caractériser l'effet du filtrage, nous définissons également l'ensemble des valeurs supprimées par le filtrage.

Définition 3 *Soient un sous-ensemble $Y \subseteq X$ tel que $|Y| = k$ et $\mathcal{A} = \{x_1 \leftarrow v_1, ..., x_k \leftarrow v_k\}$ une affectation sur Y. L'ensemble des valeurs supprimées de $\mathcal{P}(X - Y)$ par le filtrage résultant de l'affectation $\mathcal{A}$ est $\mathcal{F}_{\mathcal{A}}(X - Y) = \{(x_i, v) \in (X - Y) \times (d_i - d_i^{\mathcal{A}})\}$.*

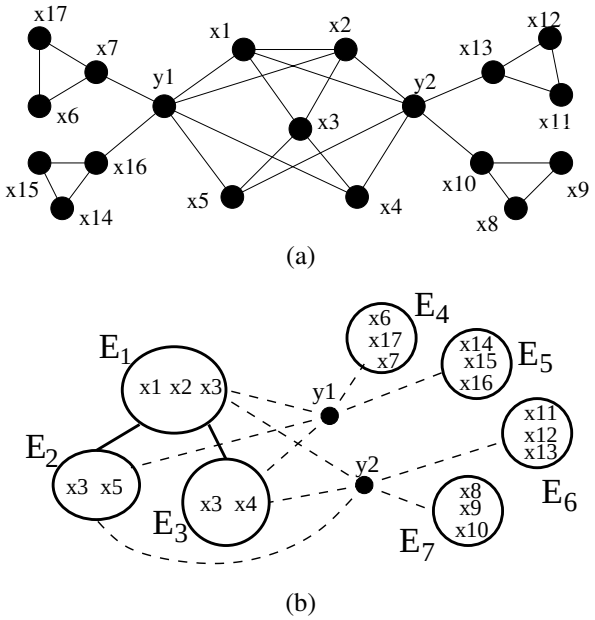


FIG. 1 – (a) Un graphe de contraintes, (b) un exemple de décomposition arborescente avec les clusters $E_1, \ldots, E_7$ et de coupe-cycle $\{y_1, y_2\}$ pour ce graphe.

Nous raffinons ensuite la définition 1 en introduisant la notion de problème filtré.

Définition 4 *Le problème filtré $\mathcal{P}_{\mathcal{A}}(X-Y)$ correspond au CSP $(X-Y, D^{\mathcal{A}}_{X-Y}, C_{X-Y}, R^{\mathcal{A}}_{X-Y})$ avec $D^{\mathcal{A}}_{X-Y} = \{d_i^{\mathcal{A}} | x_i \in X-Y\}$ et $R^{\mathcal{A}}_{X-Y} = \{r_c^{\mathcal{A}} = r_c \cap (d_j^{\mathcal{A}} \times d_k^{\mathcal{A}}) | c = \{x_j, x_k\} \in C_{X-Y}$ et $r_c \in R\}$.*

Nous pouvons constater que le filtrage effectué par FC ne modifie en rien la structure définie par le graphe de contraintes d'un problème. Il en est de même pour les décompositions arborescentes :

Propriété 1 *Une décomposition arborescente de $\mathcal{P}(X-Y)$ est une décomposition arborescente de $\mathcal{P}_{\mathcal{A}_1}(X-Y)$, et réciproquement.*

Preuve : Une décomposition arborescente ne dépend que du graphe considéré. Dans la mesure où la structure de $\mathcal{P}(X-Y)$ et celle de $\mathcal{P}_{\mathcal{A}_1}(X-Y)$ sont représentées par le même graphe de contraintes, ces deux problèmes possèdent nécessairement les mêmes décompositions arborescentes. □

A présent, nous mesurons l'impact du filtrage sur les domaines et les relations d'un problème donné au travers de la propriété suivante.

Propriété 2 *Si $\mathcal{F}_{\mathcal{A}_1}(Z) \subseteq \mathcal{F}_{\mathcal{A}_2}(Z)$, alors $\forall z_i \in Z$, $d_i^{\mathcal{A}_2} \subseteq d_i^{\mathcal{A}_1}$ et $\forall c_{jk} \in C_Z$, $r_{c_{jk}}^{\mathcal{A}_2} \subseteq r_{c_{jk}}^{\mathcal{A}_1}$.*

Preuve : Pour chaque couple $(z_i, v_i) \in \mathcal{F}_{A_1}(Z)$, le filtrage consiste à supprimer la valeur v_i du domaine de la variable z_i. Donc $\forall d_i^{\mathcal{A}_1} \in D^{\mathcal{A}_1}_{X-Y}$, $d_i^{\mathcal{A}_1} = d_i - \{v_i \in d_i | (z_i, v_i) \in \mathcal{F}_{\mathcal{A}_1}(Z)\}$. De même, nous avons $\forall d_i^{\mathcal{A}_2} \in D^{\mathcal{A}_2}_{X-Y}$, $d_i^{\mathcal{A}_2} = d_i - \{v_i \in d_i | (z_i, v_i) \in \mathcal{F}_{\mathcal{A}_2}(Z)\}$. Toutefois, comme $\mathcal{F}_{\mathcal{A}_1}(Z) \subseteq \mathcal{F}_{\mathcal{A}_2}(Z)$, nous avons $\{v_i \in d_i | (z_i, v_i) \in \mathcal{F}_{\mathcal{A}_1}(Z)\} \subseteq \{v_i \in d_i | (z_i, v_i) \in \mathcal{F}_{\mathcal{A}_2}(Z)\}$. Il en résulte que $d_i^{\mathcal{A}_2} \subseteq d_i^{\mathcal{A}_1}$.

Soient $c_{jk} \in C_Z$ une contrainte entre deux variables $x_j, x_k \in Z$ et $r_{c_{jk}}^{\mathcal{A}_1}$ et $r_{c_{jk}}^{\mathcal{A}_2}$ les relations associées obtenues consécutivement au filtrage résultant respectivement de $\mathcal{A}_1$ et de $\mathcal{A}_2$. D'après la définition 4, nous avons $r_{c_{jk}}^{\mathcal{A}_1} = r_{c_{jk}} \cap (d_j^{\mathcal{A}_1} \times d_k^{\mathcal{A}_1})$ et $r_{c_{jk}}^{\mathcal{A}_2} = r_{c_{jk}} \cap (d_j^{\mathcal{A}_2} \times d_k^{\mathcal{A}_2})$. De plus, $d_j^{\mathcal{A}_2} \times d_k^{\mathcal{A}_2} \subseteq d_j^{\mathcal{A}_1} \times d_k^{\mathcal{A}_1}$ puisque $\forall z_i \in Z$, $d_i^{\mathcal{A}_2} \subseteq d_i^{\mathcal{A}_1}$. Donc, $r_{c_{jk}}^{\mathcal{A}_2} \subseteq r_{c_{jk}}^{\mathcal{A}_1}$. □

Nous comparons maintenant les ensembles de solutions de deux sous-problèmes induits par le même sous-ensemble de variables mais résultant de filtrages différents.

Propriété 3 *Si $\mathcal{F}_{\mathcal{A}_1}(Z) \subseteq \mathcal{F}_{\mathcal{A}_2}(Z)$, alors $Sol(\mathcal{P}_{\mathcal{A}_2}(Z)) \subseteq Sol(\mathcal{P}_{\mathcal{A}_1}(Z))$ et $|Sol(\mathcal{P}_{\mathcal{A}_2}(Z))| \leq |Sol(\mathcal{P}_{\mathcal{A}_1}(Z))|$.*

Preuve : Soit $\mathcal{S}$ une solution de $\mathcal{P}_{\mathcal{A}_2}(Z)$. Montrons que $\mathcal{S}$ est aussi une solution de $\mathcal{P}_{\mathcal{A}_1}(Z)$. Par définition, $\mathcal{S}$ est une affectation consistante de toutes les variables de Z telle que $\forall c_{jk} \in C_Z$, $\mathcal{S}[c_{jk}] \in r_{c_{jk}}^{\mathcal{A}_2}$. Or, $\mathcal{F}_{\mathcal{A}_1}(Z) \subseteq \mathcal{F}_{\mathcal{A}_2}(Z)$ et, d'après la propriété 2, $\forall c_{jk} \in C_Z$, $r_{c_{jk}}^{\mathcal{A}_2} \subseteq r_{c_{jk}}^{\mathcal{A}_1}$. Par conséquent, $\forall c_{jk} \in C_Z$, $\mathcal{S}[c_{jk}] \in r_{c_{jk}}^{\mathcal{A}_1}$. Donc, $\mathcal{S}$ est bien une solution de $\mathcal{P}_{\mathcal{A}_1}(Z)$. D'où $Sol(\mathcal{P}_{\mathcal{A}_2}(Z)) \subseteq Sol(\mathcal{P}_{\mathcal{A}_1}(Z))$ et $|Sol(\mathcal{P}_{\mathcal{A}_2}(Z))| \leq |Sol(\mathcal{P}_{\mathcal{A}_1}(Z))|$. □

Le corollaire suivant précise ce qu'il en est dans le cas particulier d'une affectation $\mathcal{A}_2$ extension d'une affectation $\mathcal{A}_1$.

Corollaire 1 *Si $\mathcal{A}_2[Y_1] = \mathcal{A}_1$, alors $Sol(\mathcal{P}_{\mathcal{A}_2}(Z)) \subseteq Sol(\mathcal{P}_{\mathcal{A}_1}(Z))$ et $|Sol(\mathcal{P}_{\mathcal{A}_2}(Z))| \leq |Sol(\mathcal{P}_{\mathcal{A}_1}(Z))|$.*

Preuve : Il suffit d'observer que l'hypothèse $\mathcal{A}_2[Y_1] = \mathcal{A}_1$ implique que $\mathcal{F}_{\mathcal{A}_1}(Z) \subseteq \mathcal{F}_{\mathcal{A}_2}(Z)$. Par conséquent, en exploitant la propriété 3, on obtient $Sol(\mathcal{P}_{\mathcal{A}_2}(Z)) \subseteq Sol(\mathcal{P}_{\mathcal{A}_1}(Z))$ et $|Sol(\mathcal{P}_{\mathcal{A}_2}(Z))| \leq |Sol(\mathcal{P}_{\mathcal{A}_1}(Z))|$. □

Nous rappelons, dans la définition suivante, la notion de goods et de nogoods structurels [8].

Définition 5 *Etant donnés deux clusters E_i et E_j avec E_j un fils de E_i, un **good** (respectivement un **nogood**) structurel de E_i par rapport à E_j est une affectation consistante $\mathcal{A}$ sur $S_j = E_i \cap E_j$ telle que $\mathcal{A}$ peut (resp. ne peut pas) être étendue en une affectation consistante sur $Desc(E_j)$.*

Nous pouvons désormais caractériser les cas pour lesquels les (no)goods structurels d'un sous-problème $\mathcal{P}(X-Y)$ restent valides si nous changeons d'affectation sur Y.

Théorème 1 *Si $\mathcal{F}_{\mathcal{A}_1}(X-Y) \subseteq \mathcal{F}_{\mathcal{A}_2}(X-Y)$ et $ng(S_j)$ est un nogood du sous-problème $\mathcal{P}_{\mathcal{A}_1}(X-Y)$, alors $ng(S_j)$ est aussi un nogood pour le sous-problème $\mathcal{P}_{\mathcal{A}_2}(X-Y)$.*

Preuve : TD est une décomposition arborescente associée aux CSP $\mathcal{P}_{\mathcal{A}_1}$ et $\mathcal{P}_{\mathcal{A}_2}$. De plus, $S_j = E_i \cap E_j$ est un séparateur de TD entre le cluster E_i et un de ses fils E_j. Nous savons que $ng(S_j)$ est un nogood pour $\mathcal{P}_{\mathcal{A}_1}(X-Y)$. Donc $|Sol(\mathcal{P}_{\mathcal{A}_1}(Desc(E_j)))| = 0$ si l'affectation sur S_j correspond à ce nogood. Cependant, $\mathcal{F}_{\mathcal{A}_1}(X-Y) \subseteq \mathcal{F}_{\mathcal{A}_2}(X-Y)$ et $Desc(E_j) \subseteq X-Y$. Par conséquent, nous pouvons appliquer la propriété 3. Il en découle que $|Sol(\mathcal{P}_{\mathcal{A}_2}(Desc(E_j)))| \leq |Sol(\mathcal{P}_{\mathcal{A}_1}(Desc(E_j)))|$ et donc que $|Sol(\mathcal{P}_{\mathcal{A}_2}(Desc(E_j)))| = 0$. Autrement dit, $ng(S_j)$ est aussi un nogood pour $\mathcal{P}_{\mathcal{A}_2}(X-Y)$. □

Le théorème précédent pose une condition (inclusion) sur l'ensemble des valeurs supprimées par le filtrage pour pouvoir conclure à la validité ou non d'un nogood déjà mémorisé. Cependant, d'un point de vue algorithmique et pratique, exploiter ce théorème semble conduire à un test trop coûteux à la fois en temps et en espace. Aussi, dans le corollaire ci-dessous, nous proposons une restriction sur les filtrages résultant de deux affectations.

Corollaire 2 *Si $\mathcal{A}_2[Y_1] = A_1$ et $ng(S_j)$ est un nogood pour le sous-problème $\mathcal{P}_{\mathcal{A}_1}(X-Y)$, alors $ng(S_j)$ est aussi un nogood pour le sous-problème $\mathcal{P}_{\mathcal{A}_2}(X-Y)$.*

Preuve : En observant que $\mathcal{A}_2[Y_1] = \mathcal{A}_1$ implique que $\mathcal{F}_{\mathcal{A}_1}(Z) \subseteq \mathcal{F}_{\mathcal{A}_2}(Z)$, il suffit d'appliquer le théorème 1 pour obtenir le résultat souhaité. □

Ensuite, nous nous intéressons à conserver la validité des goods.

Théorème 2 *Si $\mathcal{F}_{\mathcal{A}_2}(Desc(E_j)) \subseteq \mathcal{F}_{\mathcal{A}_1}(Desc(E_j))$ et $g(S_j)$ est un good pour le sous-problème $\mathcal{P}_{\mathcal{A}_1}(X-Y)$, alors $g(S_j)$ est aussi un good pour le sous-problème $\mathcal{P}_{\mathcal{A}_2}(X-Y)$.*

Preuve : Comme $\mathcal{F}_{\mathcal{A}_2}(Desc(E_j)) \subseteq \mathcal{F}_{\mathcal{A}_1}(Desc(E_j))$ et $Desc(E_j) \subseteq X-Y$, nous pouvons appliquer la propriété 3. Par conséquent, $Sol(\mathcal{P}_{\mathcal{A}_1}(Desc(E_j)) \subseteq Sol(\mathcal{P}_{\mathcal{A}_2}(Desc(E_j)))$. De plus, nous savons que $g(S_j)$ est un good pour le sous-problème $\mathcal{P}_{\mathcal{A}_1}(X-Y)$. Donc, $\mathcal{P}_{\mathcal{A}_1}(Desc(E_j))$ possède au moins une solution $\mathcal{S}$ (par définition d'un good). Donc, $\mathcal{S}$ est une solution du sous-problème $\mathcal{P}_{\mathcal{A}_2}(Desc(E_j))$ également. Par conséquent, $g(S_j)$ est un good pour le sous-problème $\mathcal{P}_{\mathcal{A}_2}(X-Y)$. □

Toutes ces propriétés et corollaires peuvent être utilisés dans le cadre de l'algorithme CC-BTD-gen afin de décider quelles sont les informations qui demeurent valides entre les différents appels à BTD. Pour cela, Y correspondra à l'ensemble coupe-cycle et donc $X-Y$ à l'ensemble des variables appartenant à la décomposition arborescente associée. Le théorème 1 permet de conclure qu'à partir de deux affectations partielles $\mathcal{A}_1$ et $\mathcal{A}_2$ sur le coupe-cycle telles que $\mathcal{A}_2$ filtre au moins les mêmes valeurs que $\mathcal{A}_1$, les nogoods mémorisées par BTD pour le sous-problème $\mathcal{P}_{\mathcal{A}_1}$ restent valides pour le sous-problème $\mathcal{P}_{\mathcal{A}_2}$. Cependant, du fait de la quantité d'espace mémoire limitée, nous ne pouvons pas nous permettre de mémoriser précisément les effets du filtrage découlant de chaque affectation partielle consistante sur le coupe-cycle. Aussi, nous exploiterons plutôt le corollaire 2, qui rend possibles l'enregistrement et la réutilisation de nogoods dans le cas où nous étendons une affectation partielle consistante sur le coupe-cycle. De même, pour la réutilisation des goods, nous nous contenterons de tester la validité des goods au moment de les utiliser afin de limiter le coût global en temps de ces tests. Dans la section suivante, nous décrivons et étudions l'algorithme CC-BTD-gen.

3 Une généralisation de Cyclic-Clustering

L'algorithme CC-BTD-gen (voir algorithme 1) repose sur l'exploitation d'un ensemble coupe-cycle et d'une décomposition arborescente du graphe de contraintes. Ce coupe-cycle et cette décomposition arborescente peuvent être calculés grâce à n'importe quelle méthode, cette dernière pouvant utiliser ou non la notion de sous-graphe triangulé induit (TIS). L'algorithme CC-BTD-gen consiste à instancier de façon consistante les variables du coupe-cycle tout en vérifiant, grâce à une version dédiée de BTD, si l'affectation courante sur le coupe-cycle peut être étendue de manière consistante sur la partie correspondant à la décomposition arborescente. Ce test pouvant s'avérer coûteux, après chaque affectation consistante d'une variable du coupe-cycle, la méthode CC-BTD-gen décide, par l'intermédiaire de la fonction heuristique $ChoiceBTD$, si elle doit ou non l'effectuer. Si BTD retourne $true$, la méthode CC-BTD-gen continue la recherche sur le coupe-cycle. Dans le cas contraire, elle essaie une nouvelle valeur (s'il en reste) pour la variable courante du coupe-cycle ou revient en arrière sur la variable précédente. Ce processus est itéré jusqu'à l'obtention d'une solution (c'est-à-dire d'une affectation consistante du coupe-cycle qui peut être étendue de manière consistante sur la partie correspondant à la décomposition arborescente) ou jusqu'à ce que l'ensemble de l'espace de recherche ait été exploré.

En premier lieu, afin de pouvoir réutiliser les (no)goods enregistrés lors des différents appels à BTD, nous propo-

Algorithme 1 : CC-BTD-gen($in : \mathcal{A}, V, NG_p, in/out : G_p$)

```
1  Cons ← true
2  if ChoiceBTD(V) or V = ∅ then
3      G ← ∅ ; NG ← ∅
4      Cons ← BTD-gen(∅, E_1, V_{E_1}, NG_p, G_p, NG, G)
5      G_p ← G_p ∪ G ; NG_p ← NG_p ∪ NG
6  if Cons and V ≠ ∅ then
7      Choose x_i ∈ V ; d_i ← D_i ; Cons ← false
8      while d_i ≠ ∅ and ¬Cons do
9          Choose v ∈ d_i ; d_i ← d_i − {v}
10         if Filtering(A ∪ {x_i ← v}, x_i) then
11             Cons ←
               CC-BTD-gen(A ∪ {x_i ← v}, V − {x_i}, NG_p, G_p)
12         Unfiltering(A, x_i)
13 return Cons
```

Algorithme 2 : BTD-gen($in : \mathcal{A}, E_i, V_{E_i}, NG_p, G_p, in/out : NG, G$)

```
1  if V_{E_i} = ∅ then
2      Cons ← true ; F ← Sons(E_i)
3      while F ≠ ∅ and Cons do
4          Choose E_j ∈ F ; F ← F − {E_j}
5          S_j ← E_i ∩ E_j ;
6          if A[S_j] is a nogood into NG then Cons ← false
7          else if A[S_j] is a nogood into NG_p then Cons ← false
8          else if A[S_j] is a good into G then Cons ← true
9          else if A[S_j] is a good into G_p and CheckGood(E_j, A[S_j])
           then Cons ← true
10         else
11             Cons ←
               BTD-gen(A, E_j, E_j \(E_j ∩ E_i), NG_p, G_p, NG, G)
12             if Cons then Save the good A[S_j] into G
13             else Save the nogood A[S_j] into NG
14 else
15     Choose x_k ∈ V_{E_i} ; d_k ← D_k ; Cons ← false
16     while d_k ≠ ∅ et ¬Cons do
17         Choose w ∈ d_k ; d_k ← d_k − {w}
18         if A ∪ {x_k ← w} satisfies each constraint then
19             Cons ←BTD-gen(A ∪ {x_k ← w}, E_i, V_{E_i} − {x_k},
               NG_p, G_p, NG, G)
20 return Cons
```

Algorithme 3 : CheckGood(E_i, g)

```
1  Let S be the assignment g and its recorded extension on E_i
2  forall y ∈ E_i do
3      if S[y] ∉ d_y then return false
4  ValidGood ← true ; F ← Sons(E_i)
5  while F ≠ ∅ et ValidGood do
6      Choose E_j ∈ F ; F ← F − {E_j}
7      g_F ← good on E_j such that g_F[E_i ∩ E_j] = S[E_i ∩ E_j]
8      ValidGood ←CheckGood(E_j, g_F)
9  return ValidGood
```

sons une variante de BTD, appelée BTD-gen (voir algorithme 2), qui met en œuvre les propriétés mises en évidence dans la section précédente. BTD-gen ne se distingue de BTD que par sa capacité à exploiter des (no)goods mémorisés durant des appels antérieurs à BTD-gen. Par conséquent, cette variante est dotée de deux paramètres supplémentaires, à savoir l'ensemble G_p des goods et l'ensemble NG_p des nogoods qui ont été enregistrés lors des exécutions précédentes de BTD-gen. Quant aux ensembles G et NG, ils désignent respectivement l'ensemble des goods et celui des nogoods produits durant l'appel courant à BTD-gen. Dans la mesure où nous conservons l'intégralité des goods produits, certains d'entre eux peuvent ne pas demeurer valides dans certains appels à BTD-gen. Par conséquent, avant de réutiliser un tel good, BTD-gen doit préalablement s'assurer de sa validité vis-à-vis du sous-problème courant afin de respecter le théorème 2. Ce test est accompli par le biais de la fonction $CheckGood$ (voir algorithme 3). Cette fonction renvoie $true$ si chacune des variables de la descendance de E_i peut être instanciée avec la valeur qu'elle possédait au moment où le good g a été mémorisé. Afin de pouvoir tester facilement cette propriété, nous avons besoin de mémoriser l'extension du good sur l'ensemble des autres variables du cluster. Comme BTD, BTD-gen retourne la consistance du sous-problème associé à la décomposition arborescente TD et enraciné en E_i.

Cette version dédiée de BTD est exploitée dans CC-BTD-gen pour vérifier que l'affectation partielle courante sur le coupe-cycle peut être étendue de façon consistante sur la partie correspondant à la décomposition arborescente. Si BTD-gen($\emptyset, E_1, V_{E_1}, NG_p, G_p, NG, G$) renvoie $false$, alors CC-BTD-gen tente d'instancier la variable courante du coupe-cycle avec une nouvelle valeur (s'il en reste). En l'absence d'autres valeurs, CC-BTD-gen revient en arrière sur la variable précédente. Dans le cas où l'appel à BTD-gen renvoie $true$, CC-BTD-gen continue la recherche en instanciant une nouvelle variable du coupe-cycle. Dans les deux cas, l'ensemble G des goods découverts lors de l'appel à BTD-gen est ajouté à l'ensemble G_p. Le processus est similaire pour les nogoods, si ce n'est que NG_p ne peut être modifié au delà de l'appel courant à CC-BTD-gen. Autrement dit, à chaque retour en arrière d'un appel à CC-BTD-gen, nous oublions les nogoods mémorisés durant cet appel, afin de respecter le corollaire 2.

Enfin, la fonction booléenne $ChoiceBTD$ définit, après chaque affectation d'une variable du coupe-cycle, si BTD-gen doit être appelé ou non. Si elle retourne $false$ et qu'il reste des variables non instanciées dans le coupe-cycle, CC-BTD-gen va essayer d'en affecter une avec l'algorithme Forward-Checking (lignes 7-12). Si $ChoiceBTD$ renvoie $true$, CC-BTD-gen fait appel à BTD-gen et conserve les nouveaux goods et nogoods mémorisés (lignes 3-5). Notons que cette heuristique peut être entièrement dynamique. Elle peut donc décider d'appeler BTD-gen après n'importe quelle affectation partielle consistante du coupe-cycle.

Nous illustrons maintenant l'algorithme CC-BTD-gen avec un exemple. Considérons, pour cela, le graphe de contraintes de la figure 1(a) et une décomposition arborescente avec cinq composantes connexes et un coupe-cycle contenant deux variables y_1 et y_2 comme représenté dans la figure 1(b). Supposons que CC-BTD-gen commence par instancier des variables du coupe-cycle. Par exemple, s'il

a affecté seulement la variable y_1, le filtrage de FC qui s'ensuit a pu réduire le domaine des variables non instanciées voisines de y_1, à savoir x_1, x_2, x_4, x_5, x_7 et x_{16}. Ensuite, l'heuristique $ChoiceBTD$ peut décider de résoudre le sous-problème associé à la décomposition arborescente avec BTD-gen. Si BTD-gen retourne $false$, alors CC-BTD-gen va modifier l'affectation de y_1. Sinon, l'algorithme CC-BTD-gen va tenter d'instancier de manière consistante la variable y_2 du coupe-cycle. S'il y parvient, un nouvel appel à BTD-gen est accompli puisque toutes les variables du coupe-cycle sont instanciées. Si BTD-gen renvoie $true$, alors le CSP est consistant. Dans le cas contraire, CC-BTD-gen va essayer une nouvelle valeur pour y_2, et s'il n'en reste plus va revenir sur y_1.

Théorème 3 *BTD-gen est correct, complet et termine.*

Preuve : BTD est correct, complet et termine [8]. Comme BTD-gen ne se distingue de BTD que par l'exploitation des (no)goods enregistrés durant les appels précédents à BTD-gen, nous devons simplement prouver que l'utilisation de ces (no)goods ne remet pas en cause la validité, la complétude et la terminaison de BTD. Si Y désigne le coupe-cycle, $X-Y$ correspond aux variables de la décomposition arborescente. D'après le corollaire 2, si $\mathcal{A}$ est une affectation consistante sur Y, alors chaque nogood pour le sous-problème $\mathcal{P}_{\mathcal{A}}(X-Y)$ est également un nogood pour le sous-problème $\mathcal{P}_{\mathcal{A}'}(X-Y)$ avec $\mathcal{A}'$ une extension consistante de $\mathcal{A}$ sur le coupe-cycle. De plus, lorsqu'on revient en arrière de $\mathcal{A}' = \mathcal{A} \cup \{x_k \leftarrow w\}$ vers $\mathcal{A}$, CC-BTD-gen oublie tous les nogoods qui ont été mémorisés depuis l'affectation de la variable x_k à la valeur w. Par conséquent, le corollaire 2 est bien respecté et il est bien valide d'utiliser les nogoods de NG_p. Concernant l'exploitation des goods de G_p, si la fonction $CheckGood$ renvoie $true$ pour un good donné, il s'ensuit qu'utiliser ce good est valide d'après le théorème 2. Aussi, comme l'algorithme BTD-gen n'emploie que des (no)goods valides, il est correct, complet et termine. □

Théorème 4 *CC-BTD-gen est correct, complet et termine.*

Preuve : Afin de prouver plus aisément la correction, la complétude et la terminaison de CC-BTD-gen, nous allons considérer l'algorithme avec un point de vue légèrement différent. La résolution effectuée par CC-BTD-gen peut être décomposée en deux phases. La première phase revient à résoudre la partie coupe-cycle avec une version modifiée de l'algorithme FC. Cette dernière consiste simplement à utiliser l'algorithme FC classique et, quand l'heuristique $ChoiceBTD$ renvoie $true$, à appeler BTD-gen. Cet appel à BTD-gen peut être vu comme un test de consistance supplémentaire. En effet, on vérifie si l'affectation partielle courante sur le coupe-cycle peut être étendue ou non de manière consistante sur les variables de la décomposition arborescente. La validité de cette coupe supplémentaire ne dépend bien sûr que de la validité de BTD-gen.

La seconde phase débute quand toutes les variables du coupe-cycle sont instanciées de façon consistante. Cette affectation constitue donc une solution du sous-problème lié au coupe-cycle, qui doit être étendue de manière consistante sur la partie associée à la décomposition arborescente. Cette seconde phase est accomplie à l'aide de BTD-gen.

Dans la mesure où les algorithmes FC et BTD-gen sont corrects, complets et terminent, il en est nécessairement de même pour l'algorithme CC-BTD-gen. □

Dans les deux théorèmes suivant, nous noterons n le nombre de variables du CSP, m le nombre de contraintes, d la taille du plus grand domaine, k la taille du coupe-cycle, w la largeur de la décomposition arborescente considérée, et s la taille de la plus grande intersection entre deux clusters de la décomposition arborescente.

Théorème 5 *BTD-gen a une complexité en temps en $O(n(n+m)d^{w+1})$ et une complexité en espace en $O(nwd^s)$.*

Preuve : La preuve est similaire à celle de BTD [8]. Nous devons juste prendre en compte le coût en temps additionnel nécessaire pour tester la validité des goods de G_p et l'espace supplémentaire requis pour mémoriser l'extension de chaque good sur le cluster correspondant. □

Théorème 6 *CC-BTD-gen a une complexité en temps en $O(n(n+m)d^{w+k+2})$ et une complexité en espace en $O(nwd^s)$.*

Preuve : Dans le pire des cas, CC-BTD-gen appelle BTD-gen après chaque affectation d'une variable du coupe-cycle. Comme le nombre d'affectations partielles du coupe-cycle est borné par d^{k+1}, CC-BTD-gen a une complexité en temps en $O(n(n+m)d^{w+1}.d^{k+1} + nm.d^{k+1})$, c'est-à-dire en $O(n(n+m)d^{w+k+2})$. Pour la complexité en espace, cela dépend uniquement de BTD-gen. Donc, CC-BTD-gen a une complexité en espace en $O(nwd^s)$. □

4 Résultats expérimentaux

Dans cette section, nous évaluons expérimentalement l'intérêt pratique de l'algorithme CC-BTD-gen par rapport à des méthodes structurelles classiques à savoir CC-BTD$_1$, CC-BTD$_2$ et BTD et à une méthode énumérative classique, en l'occurrence FC. Les tests sont effectués sur des problèmes structurés générés aléatoirement. Plus précisément, nous utilisons un générateur de CSP binaires. Ce générateur construit d'abord un premier sous-problème dont le graphe de contraintes est triangulé. Puis, il génère un second sous-problème qui correspond à l'ensemble coupe-cycle. Enfin, il relie les deux sous-problèmes en ajoutant

Classes $(\mathbf{n}, \mathbf{d}, \mathbf{r}, \mathbf{t_1}, \mathbf{t_2}, \mathbf{t_3}, \mathbf{s}, \mathbf{k}, \mathbf{e_1}, \mathbf{e_2})$	BTD						CC-BTD (1,2,gen)		
	min-fill		Fusion						
			$\mathbf{CC_{gen}}$		BY		BY		
	w	s	w	s	w	s	k	w	s
(a) (120, 15, 15, 65, 70, 40, 5, 15, 80, 30)	40,7	7	40,8	9,2	54,5	9,1	13,9	13,9	4,9
(b) (120, 15, 15, 65, 80, 30, 5, 15, 80, 30)	40,7	7	27,2	14,4	38,3	14,1	13,9	13,9	4,9
(c) (150, 15, 15, 65, 70, 40, 5, 15, 65, 30)	54	6	42,3	9,3	59	9,5	14,2	13,9	5
(d) (150, 15, 15, 65, 80, 20, 5, 15, 50, 30)	38,6	7	42	9,2	56,2	9,7	13,3	14	5
(e) (150, 15, 15, 64, 60, 60, 5, 15, 50, 30)	30	8	42	9,2	56,2	9,7	13,3	14	5
(f) (200, 15, 15, 64, 30, 30, 5, 15, 30, 20)	36	6	34,2	9,3	42,1	9,7	12	13,9	5

TAB. 1 – Paramètres des différentes classes et valeurs des paramètres structurels des différentes méthodes considérées. Le coupe-cycle et la décomposition arborescente associée sont calculés à partir de l'algorithme de Balas et Yu (BY) ou sont ceux produits par le générateur aléatoire (CC_{gen}).

Classes	FC		BTD				CC-BTD						CC-BTD-gen					
			min-fill		Fusion		$CC\text{-}BTD_1$		$CC\text{-}BTD_2$		H_k		H2		H1	
(a)	$>_8$	281	$>_{23}$	585	$>_{10}$	240	$>_{27}$	649	$>_1$	25.3	$>_1$	25		3,51		1,80
(b)	$>_8$	264	$>_{19}$	489		4,99		24,6		0,84		1,05		5,49		6,44
(c)	$>_8$	260	$>_{26}$	643	$>_{20}$	625	$>_{32}$	773	$>_3$	76	$>_3$	74,8		14,28		0,70
(d)	$>_5$	195	$>_{41}$	993	$>_{33}$	839	$>_{31}$	748	$>_2$	52,4	$>_2$	51,9		4,13		0,34
(e)	$>_{10}$	317	$>_{41}$	986	$>_{41}$	986	$>_{33}$	795	$>_5$	124	$>_5$	121	$>_3$	82,1		15,91
(f)	$>_{15}$	605	$>_{42}$	1008	$>_{39}$	953	$>_{34}$	816	$>_6$	144	$>_6$	144	$>_5$	120	$>_1$	24,4

TAB. 2 – Temps d'exécution (en secondes) pour les différentes méthodes sur les classes considérées. Le coupe-cycle et la décomposition arborescente associée sont ceux produits par le générateur aléatoire (CC_{gen}).

aléatoirement un certain nombre de contraintes. Dix paramètres sont nécessaires pour générer de tels problèmes, à savoir n, d, r, t_1, t_2, t_3, s, k, e_1 et e_2. L'instance générée comporte $n + k$ variables qui ont toutes un domaine de taille d. Plus précisément, la partie triangulée possède n variables réparties dans des cliques de taille au plus r et dont l'intersection entre deux cliques est de taille au plus s. La partie coupe-cycle est composée de k variables et e_1 contraintes. Enfin, e_2 contraintes relient ces deux parties entre elles. Les relations associées à chaque contrainte possèdent t_1 tuples interdits pour les contraintes entre deux variables de la partie triangulée, t_2 pour les contraintes entre deux variables du coupe-cycle et t_3 pour les contraintes liant une variable du coupe-cycle à une variable de la partie triangulée. Par conséquent, une classe de ces problèmes aléatoires est définie par la donnée de ces dix paramètres : $(n, d, r, t_1, t_2, t_3, s, k, e_1, e_2)$. Pour chaque classe considérée, le nombre de problèmes consistants est approximativement le même que celui de problèmes inconsistants. Dans nos expérimentations, BTD et BTD-gen exploitent l'algorithme FC pour résoudre chaque cluster. Concernant l'ordre d'instanciation des variables dans FC ou au sein d'un cluster pour BTD et BTD-gen, nous utilisons l'heuristique *dom/deg* qui choisit comme prochaine variable la variable x_i qui minimise le rapport $\frac{|d_{x_i}|}{|\Gamma_{x_i}|}$ avec d_{x_i} le domaine courant de de x_i et Γ_{x_i} l'ensemble des variables voisines de x_i.

Nous comparons les résultats obtenus par CC-BTD-gen avec ceux de BTD, $CC\text{-}BTD_1$ et $CC\text{-}BTD_2$ d'une part, et de FC d'autre part. Pour des raisons d'efficacité, les méthodes basées sur l'approche Cyclic-Clustering classique, comme $CC\text{-}BTD_i$, nécessitent un ensemble coupe-cycle ayant peu de solutions. Malheureusement, à notre connaissance, aucune méthode satisfaisante n'existe pour déterminer une telle structure. Nous utiliserons donc d'une part l'algorithme de Balas et Yu [2] qui permet de calculer un sous-graphe induit triangulé à partir duquel est ensuite déduit le coupe-cycle. D'autre part, nous emploierons également le coupe-cycle et la décomposition arborescente utilisés lors de la génération de l'instance. L'idée en procédant ainsi est simplement d'observer le comportement de ces algorithmes quand une structure a priori adéquate est exploitée.

Pour CC-BTD-gen, et plus particulièrement, pour la fonction $ChoiceBTD$, nous avons testé plusieurs heuristiques, notées H_i avec $i = \{1, ..., k\}$. Chaque heuristique H_i décide de résoudre la partie associée à la décomposition arborescente si, depuis le dernier appel à BTD-gen, au moins i variables du coupe-cycle ont été instanciées de façon consistante et si au moins une valeur a été supprimée par filtrage dans un domaine associé à une variable de la décomposition arborescente. Initialement, chaque heuristique effectue un appel préliminaire à BTD-gen, à l'image de l'appel initial à BTD accompli par $CC\text{-}BTD_2$. Dans ce papier, nous ne présenterons que les résultats pour les heuristiques H_1, H_2 et H_k.

Classes	BTD		CC-BTD				CC-BTD-gen					
	Fusion		CC-BTD$_1$		CC-BTD$_2$		H$_k$		H2		H1	
(a)	$>_{21}$	574	$>_{32}$	792	$>_6$	168	$>_6$	157	$>_3$	107	$>_1$	28,2
(b)	$>_9$	297	$>_{22}$	617	$>_7$	186	$>_5$	155	$>_1$	41,3	$>_1$	35,5
(c)	$>_{28}$	712	$>_{41}$	1016	$>_{14}$	339	$>_{13}$	313	$>_{12}$	289	$>_6$	197
(d)	$>_{31}$	771	$>_{36}$	877	$>_7$	181	$>_7$	177	$>_4$	106	$>_4$	96,3
(e)	$>_{36}$	881	$>_{33}$	795	$>_7$	171	$>_7$	170	$>_4$	125	$>_3$	77,4
(f)	$>_{40}$	981	$>_{39}$	936	$>_{13}$	312	$>_{13}$	312	$>_{11}$	264	$>_{10}$	240

TAB. 3 – Temps d'exécution (en secondes) pour les différentes méthodes sur les classes considérées. Le coupe-cycle et la décomposition arborescente associée sont calculés à partir de l'algorithme de Balas et Yu.

Les expérimentations sont effectuées sur un PC sous Linux doté d'un Pentium IV 3.2 GHz d'Intel et d'un 1 Go de mémoire vive. Les temps d'exécution sont exprimés en secondes. Pour chaque classe, nous lançons la résolution sur 50 instances. Les résultats présentés pour chaque classe sont des moyennes des résultats obtenus sur ces 50 instances. La notation $>_i$ indique que i instances n'ont pu être résolues par l'algorithme correspondant dans le temps imparti (à savoir 20 minutes). Dans un tel cas, comme le temps de résolution réel n'est pas connu, nous ajoutons une pénalité de 20 minutes par instance non résolue.

Dans un premier temps, nous pouvons observer une augmentation importante du temps d'exécution quand on passe de l'heuristique H_1 à l'heuristique H_k (excepté pour la seconde classe). En particulier, si H_1 et H_2 sont relativement proches conceptuellement, en pratique CC-BTD-gen a un comportement nettement plus performant avec H_1 qu'avec H_2. D'un point de vue théorique, dans le pire des cas, si $i < j$, alors H_i est susceptible d'appeler BTD-gen plus souvent que H_j. Cependant, en pratique, nous avons constaté que souvent, tester la consistance de la partie associée à la décomposition arborescente permet d'élaguer significativement l'espace de recherche relatif à la partie coupe-cycle du problème. Pour cette raison, H_1 obtient les meilleurs résultats par rapport aux autres heuristiques H_i, mais aussi aux méthodes CC-BTD$_1$ et CC-BTD$_2$. CC-BTD-gen avec H_k et CC-BTD$_2$ sont conceptuellement très proches. En effet, ces deux méthodes instancient intégralement le coupe-cycle avant de résoudre la partie correspondant à la décomposition arborescente. La seule différence notable est la réutilisation par CC-BTD-gen avec H_k des goods mémorisés lors des différents appels à BTD-gen. Cette différence se traduit généralement en pratique par des résultats meilleurs, ou dans le pire des cas comparables, pour CC-BTD-gen avec H_k par rapport à CC-BTD$_2$. Plus précisément, si, bien évidemment, les deux méthodes développent autant de nœuds sur la partie coupe-cycle, CC-BTD-gen en développe moins durant la résolution de la partie associée à la décomposition arborescente que CC-BTD$_2$, grâce aux goods mémorisés lors des appels précédents à BTD que CC-BTD-gen réutilise. Les cas où les deux méthodes sont équivalentes en temps correspondent alors souvent aux instances pour lesquelles l'apport des goods dans CC-BTD-gen n'est pas suffisant pour compenser le temps supplémentaire requis pour vérifier leur validité. Par ailleurs, comme cela a été mis en évidence dans [9], l'utilisation de nogoods mémorisés lors de l'appel préliminaire à BTD permet à CC-BTD$_2$ de résoudre plus d'instances que CC-BTD$_1$ et explique ainsi l'écart de temps important entre les deux méthodes. Concernant BTD, la quasi-totalité des instances n'ont pas une structure propice au calcul d'une décomposition arborescente par triangulation, ce qui explique les difficultés que rencontre BTD basé sur min-fill pour résoudre ces problèmes. C'est en particulier le cas, par exemple, quand le graphe associé à la partie coupe-cycle est peu dense et quand le coupe-cycle et la décomposition arborescente sont faiblement connectés. Quand la décomposition arborescente est calculée grâce à la méthode Fusion, BTD obtient des résultats meilleurs mais se révèle tout de même moins performant que CC-BTD-gen (excepté pour la classe (b)). Enfin, nous pouvons constater que FC rencontre également de grandes difficultés lors de la résolution de ces instances. En pratique, plusieurs instances de chaque classe ne sont pas résolues par FC et les temps de résolution sont globalement supérieurs à ceux obtenus par CC-BTD-gen.

Pour terminer, nous pouvons remarquer que CC-BTD$_i$ et CC-BTD-gen sont plus efficaces lorsqu'ils exploitent le coupe-cycle et la décomposition arborescente utilisés lors de la génération de chaque instance que lorsqu'ils emploient ceux produits via l'algorithme de Balas et Yu. Un tel résultat était prévisible. Il permet toutefois de souligner l'absence de méthodes pertinentes pour calculer à la fois un coupe-cycle et une décomposition arborescente de qualité vis-à-vis de la résolution, ce qui constitue un véritable problème pour de telles méthodes structurelles. Si l'algorithme de Balas et Yu permet effectivement de calculer un sous-graphe triangulé induit à partir duquel est déduit le coupe-cycle, il ne prend nullement en compte la résolution qui suivra. A titre d'exemple, pour les problèmes aléatoires que nous avons utilisés, cet algorithme obtient des valeurs de paramètres structurels (w, k et s) proches de celles utilisées pour la génération. Toutefois, en pratique, les coupe-cycle et décompositions arborescentes produites par cette

méthode conduisent à une résolution de moins bonne qualité. De plus, cet algorithme calcule souvent de grands ensembles coupe-cycle et des décompositions arborescentes triviales. Par exemple, nous l'avons testé sur des problèmes d'allocation de fréquence (à savoir les instances fapp de la dernière compétition de CSP [12]). A l'arrivée, la grande majorité des variables se trouve dans le coupe-cycle alors que la taille des clusters de la décomposition arborescente n'excède jamais 3. En plus de l'heuristique de Balas et Yu, nous avons essayé d'autres heuristiques pour calculer un coupe-cycle et une décomposition arborescente convenables (en utilisant ou non la notion de TIS). Malheureusement, à l'heure actuelle, aucune de ces heuristiques ne s'est révélée pertinente. Aussi, cela nous laisse à penser qu'une étude similaire à celle réalisée dans [7] pour les décompositions arborescente devra être menée afin d'améliorer l'efficacité pratique de ces méthodes structurelles. Dans le même temps, même si l'heuristique H_1 a fourni de bons résultats, il pourrait être intéressant de rechercher des heuristiques plus pertinentes pour la fonction $ChoiceBTD$.

5 Conclusion et discussion

Nous avons proposé une nouvelle méthode pour résoudre des CSP structurés. Cette méthode généralise et améliore l'approche Cyclic-Clustering [6]. Plus précisément, elle exploite un coupe-cycle et une décomposition arborescente dont le calcul est totalement indépendant de la notion de sous-graphe triangulé induit, ce qui apporte plus de liberté dans une phase cruciale de la méthode. Ensuite, CC-BTD-gen peut vérifier si l'affectation courante sur le coupe-cycle peut s'étendre de manière consistante sur la partie associée à la décomposition arborescente même si les variables de l'ensemble coupe-cycle ne sont pas toutes affectées. Cela permet alors à CC-BTD-gen d'avoir une vision plus globale du problème à résoudre. Enfin, CC-BTD-gen emploie une version dédiée de BTD qui met en œuvre des propriétés permettant d'exploiter des (no)goods mémorisés lors des précédents appels à BTD, évitant ainsi de nombreuses redondances dans la recherche. Les expérimentations préliminaires ont montré l'intérêt pratique de notre approche. En particulier, CC-BTD-gen s'avère significativement meilleur que CC-BTD$_i$.

Dans le cadre des CSP, peu de travaux ont été développés autour de la notion de coupe-cycle couplé avec une décomposition arborescente. [3] présente une méthode proche de Cyclic-Clustering dans laquelle la partie associée à la décomposition arborescente est résolue avec la méthode Adaptive-Consistency. De même, à notre connaissance, le calcul simultané de coupe-cycle et de décomposition arborescente pertinents en terme de résolution de CSP n'a pas encore été étudié à ce jour. Il va de soi qu'un tel travail devra être mené afin d'améliorer l'efficacité pratique de ce type d'approches, à l'image de l'étude réalisée dans [7] sur les décompositions arborescentes. En particulier, cela pourrait s'avérer très utile dans le cadre de la résolution d'instances issues du monde réel. Enfin, l'exploitation d'un coupe-cycle et d'une décomposition arborescente dynamiques pourrait se révéler fort prometteuse.

Références

[1] S. Arnborg, D. Corneil, and A. Proskuroswki. Complexity of finding embeddings in a k-tree. *SIAM Journal of Discrete Mathematics*, 8 :277–284, 1987.

[2] E. Balas and C. Yu. Finding a maximum clique in an arbitrary graph. *Siam Journal on Computing*, 15(4) :1054–1068, 1986.

[3] R. Dechter. *Constraint processing*. Morgan Kaufmann Publishers, 2003.

[4] R. Dechter and J. Pearl. Tree-Clustering for Constraint Networks. *Artificial Intelligence*, 38 :353–366, 1989.

[5] R. Haralick and G. Elliot. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14 :263–313, 1980.

[6] P. Jégou. Cyclic-Clustering : a compromise between Tree-Clustering and the Cycle-Cutset method for improving search efficiency. In *Proceedings of the European Conference on Artificial Intelligence (ECAI)*, pages 369–371, 1990.

[7] P. Jégou, S. N. Ndiaye, and C. Terrioux. Computing and exploiting tree-decompositions for solving constraint networks. In *Proc. of CP*, pages 777–781, 2005.

[8] P. Jégou and C. Terrioux. Hybrid backtracking bounded by tree-decomposition of constraint networks. *Artificial Intelligence*, 146 :43–75, 2003.

[9] P. Jégou and C. Terrioux. A Time-space Trade-off for Constraint Networks Decomposition. In *Proceedings of the IEEE International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 234–239, 2004.

[10] N. Robertson and P.D. Seymour. Graph minors II : Algorithmic aspects of treewidth. *Algorithms*, 7 :309–322, 1986.

[11] D. Sabin and E. Freuder. Contradicting Conventional Wisdom in Constraint Satisfaction. In *Proceedings of the European Conference on Artificial Intelligence (ECAI)*, pages 125–129, 1994.

[12] M. R. C. van Dongen, C. Lecoutre, and O. Roussel, editors. *Third International CSP Solver Competition*, 2008. `http://cpai.ucc.ie/08/`.

Poupées russes et décomposition arborescente

M. Sanchez, D. Allouche, S. de Givry, T. Schiex
UBIA, UR 875, INRA, F-31320 Castanet Tolosan, France
{msanchez,allouche,degivry,tschiex}@toulouse.inra.fr

Abstract

Les problèmes d'optimisation dans les modèles graphiques ont été étudiés dans différents formalismes de l'intelligence artificielle tels que les CSP pondérés (weighted CSP ou WCSP), le maximum de satisfiabilité (MaxSAT) ou dans les réseaux probabilistes (réseaux Bayésiens et champs markoviens). En identifiant des sous-problèmes conditionnellement indépendants,qui sont résolus de façon indépendante et dont l'optimum est mémorisé, il est possible de rendre les algorithmes de type Séparation-Évaluation (Branch and Bound) asymptotiquement plus efficaces. Mais la localité des bornes induites par la décomposition affaiblit les effets réels de ce résultat asymptotique car de nombreux sous-problèmes, ne participant pas à la construction d'une solution optimale, sont résolus à l'optimum inutilement.

En s'inspirant de l'algorithme des poupées russes (RDS pour Russian Doll Search), une approche possible pour surmonter cette faiblesse consiste à (récursivement) résoudre une relaxation de chacun des sous-problèmes pour obtenir des bornes renforcées. L'algorithme ainsi défini généralise à la fois l'algorithme RDS mais aussi les algorithmes de types Branch and Bound exploitant une décomposition arborescente tels que BTD ou AND-OR Branch and Bound. Nous étudions son efficacité sur différents problèmes et fermons une instance très dure de problème d'affectation de fréquences ouverte depuis plus de 10 ans.

1 Introduction

Le traitement de modèles graphiques est un problème central de l'intelligence artificielle. L'optimisation d'une combinaison de fonctions de coût local est en particulier étudiée dans les réseaux de contraintes valués [21] . Elle permets de capturer des problèmes tels que le problème MaxSAT pondéré, CSP pondérés, recherche d'une explication de probabilité maximum (MPE pour Maximum Probability Explanation) dans les réseaux Bayésiens et les champs markoviens. Ses applications directes sont nombreuses notamment en *affectation de ressources* [23, 2] et en bioinformatique [19].

Ces dernières années, afin de résoudre des problèmes de satisfaction, d'optimisation ou de comptage, différents algorithmes de recherche arborescente exploitant le couplage d'une décomposition arborescente du problème avec une propagation des informations dures du réseau ont été proposés. Cette classe d'algorithme couvre l'algorithme de Recursive Conditioning [7], l'algorithme Backtrack bounded by Tree Decomposition (BTD) [22] et l'algorithme AND-OR graph search [17], tous héritiers de l'algorithme de Pseudo-Tree Search [11]. En comparaison avec des algorithmes de recherche arborescente traditionnels, ces algorithmes offrent des complexités asymptotiques améliorées, seulement exponentielles dans la largeur d'arbre du graphe du problème. Ce gain s'obtient toutefois au prix d'une restriction dans l'ordre d'affectation des variables (voir [13]).

Dans le contexte de l'optimisation (sans perte de généralité, nous considérons des problèmes de minimisation), l'exploitation d'une décomposition du problème a pour effet secondaire la perte de bornes globales. Dans les algorithmes Branch and Bound en profondeur d'abord traditionnels (DFBB pour Depth First Branch and , Bound), l'élagage se produit dès que le coût de la meilleure solution connue (le majorant) rencontre un minorant spécifique, habituellement calculé à partir d'une relaxation du problème global. Lorsqu'une décomposition arborescente est exploitée et qu'une affectation des variables connectant un sous-problème au reste du problème rend ce sous-problème indépendant, une solution optimale du sous-problème (conditionné par cette affectation) est recherchée. Ce calcul de solution optimale est coûteux, et souvent inutile car la solution optimale du sous-problème obtenue ne fait pas nécessairement partie de la solution optimale globale (du fait du reste du problème). Pour limiter ce comportement de "trashing" désagréable, il est possible

d'informer l'algorithme en charge de la recherche d'une solution optimale qu'il doit absolument produire une solution suffisamment bonne pour pouvoir s'intégrer dans une solution globale ou sinon s'arrêter dès qu'il prouve que c'est infaisable. Ce *majorant initial* est obtenu en soustrayant du majorant global une combinaison des minorants disponibles sur le reste du problème.

Dans cet article, nous avons tenté d'améliorer l'algorithme décrit dans [9] qui introduit des minorants obtenus par un filtrage *cohérence locale pondérée* [6] dans l'algorithme BTD. Dans l'objectif d'une meilleure prise en compte de la contribution du problème global dans la définition des majorants locaux, nous remplaçons le moteur de type "Branch and Bound" utilisé dans BTD par une algorithme de type "Poupées Russes" [23]. L'algorithme RDS est étendu afin de pouvoir exploiter une décomposition arborescente et un niveau de cohérence local plus élevé. Dans l'algorithme RDS-BTD ainsi défini, des relaxations des sous-problèmes conditionnellement indépendants identifiés par la décomposition arborescente sont résolues de façon récursive. Les coûts des solutions optimales de ces relaxations définissent de puissants minorants et fournissent donc des majorants locaux améliorés, permettant ainsi de réduire de façon drastique le comportement de "trashing" observé.

Dans la suite de l'article, nous introduisons d'abord le cadre des CSP pondérés, l'algorithme BTD et l'algorithme RDS. Nous présentons ensuite notre algorithme générique qui peut se spécialiser en BTD ou en RDS. Enfin, l'algorithme RDS-BTD est évalué sur différentes instances de problèmes réels issues du domaine de l'affectation de fréquences et de la sélection de "Tag SNP" (en bioinformatique).

2 Weighted Constraint Satisfaction Problem

Un CSP pondéré (Weighted CSP, WCSP) est un quadruplet (X, D, W, m). X et D sont des ensembles de n variables et domaines, comme dans les CSP classiques. Le domaine de la variable i est noté D_i, avec une taille de domaine maximum notée d. Étant donné un sous-ensemble de variables $S \subseteq X$, on note $\ell(S)$ l'ensemble des n-uplets (tuples) définis sur S. W est un ensemble de fonctions de coût. Chaque fonction de coût (ou contrainte molle) w_S de W est définie sur un ensemble de variables S appelé sa portée. Cette dernière est supposée différent pour chaque fonction de coût. Une fonction de coût w_S assigne un coût à chaque affectation des variables de S i.e. : $w_S : \ell(S) \rightarrow [0, m]$. L'ensemble des coûts possibles est $[0, m]$ où $m \in \{1, \ldots, +\infty\}$ représente un coût intolérable. Les coûts sont combinés par l'addition bornée $\oplus$, définie par $a \oplus b = \min\{m, a+b\}$, et comparés entre eux via $\geq$. Notez que le coût intolérable m peut être fini ou infini et qu'un coût b peut être soustrait d'un coût a plus large en utilisant l'opération $\ominus$ où $a \ominus b$ est égal à $(a - b)$ si $a \neq m$. Sinon, il est égal à m.

Pour les fonctions de coût binaires et unaires, nous utilisons des notations simplifiées : une fonction de coût binaire entre les variables i et j est notée w_{ij}. Une fonction de coût unaire sur la variable i est notée w_i. Nous faisons l'hypothèse qu'il existe pour chaque variable, une fonction de coût unaire w_i ainsi qu'une fonction de coût d'arité nulle notée $w_\varnothing$ correspondant à un coût constant payé par toutes affectations.

Le coût d'une affectation complète $t \in \ell(X)$ dans un problème $P = (X, D, W, m)$ est égal à $Val_P(t) = \bigoplus_{w_S \in W} w_S(t[S])$ où $t[S]$ représente la projection habituelle d'un tuple sur l'ensemble de variables S. La minimisation de $Val_P(t)$ défini un problème d'optimisation avec un problème de décision associé qui est NP-complet.

L'établissement d'une cohérence locale donnée sur un problème P consiste à transformer $P = (X, D, W, m)$ en un problème $P' = (X, D, W', m)$ qui est équivalent à P ($Val_P = Val_{P'}$) et qui satisfait la propriété de cohérence locale considérée. Ce processus peut augmenter $w_\varnothing$ et fournir ainsi un minorant amélioré du coût optimum. Il s'appuie sur l'application de transformations préservant l'équivalence (EPT) qui déplace les coûts entre des portées différentes [20, 6, 10, 5].

3 Décompositions arborescentes et algorithme DFBB

Une décomposition arborescente d'un WCSP connexe est définie par un arbre (C, T). Chaque élément C_e de l'ensemble $C = \{C_1, \ldots, C_k\}$ des sommets de l'arbre est appelé un "cluster". Il est formé d'un sous-ensemble de variables ($C_e \subset X$). T est un ensemble d'arêtes définissant un graphe acyclique connexe (un arbre) sur C. L'ensemble des clusters C doit couvrir toutes les variables ($\bigcup_{C_e \in C} C_e = X$) et toutes les fonctions de coût ($\forall w_S \in W, \exists C_e \in C$ *t.q.* $S \subset C_e$). De plus, si une variable i apparaît dans deux clusters C_e et C_g, i doit aussi apparaître dans tout cluster C_f sur l'unique chemin reliant C_e et C_g dans T.

Pour un WCSP donné, nous considérons une décomposition arborescente enracinée (C, T) définie par une racine notée C_1. Dans T, le père (resp. les fils) d'un cluster C_e est noté $Père(C_e)$ (resp. $Fils(C_e)$). Le séparateur de C_e est l'ensemble $S_e = C_e \cap Père(C_e)$. Les variables *propres* d'un cluster C_e est l'ensemble $V_e = C_e \setminus S_e$.

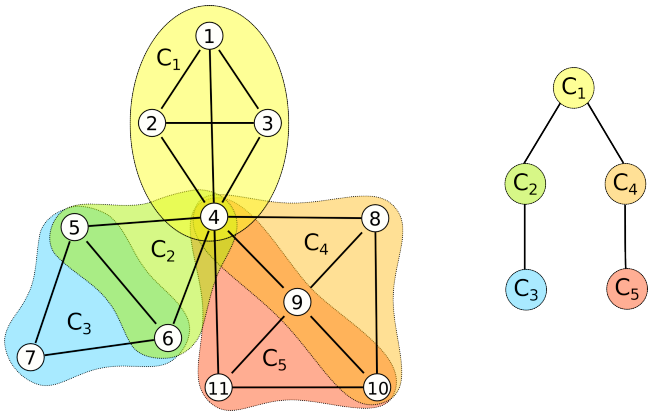


FIG. 1 – Le graphe d'un WCSP et une décomposition arborescente.

La propriété essentielle d'une décomposition arborescente est que l'affectation d'un séparateur S_e coupe le problème initial en (au moins) deux sous-problèmes qui peuvent être résolus indépendemment l'un de l'autre. Le premier sous problème, noté P_e, est défini par les variables de C_e ainsi que celles de tous les clusters descendants dans T. Ses fonctions de coût sont celles qui impliquent *au moins* une variable propre de ces clusters. Les fonctions de coût restantes, avec les variables qu'elles impliquent forment le sous-problème restant.

Exemple 1 *Considérons le MaxCSP de la figure 1. Il a onze variables avec deux valeurs (a, b) par domaine. Les arêtes représentent des contraintes de différence binaires entre les deux variables ($w_{ij}(a, a) = w_{ij}(b, b) = 1, w_{ij}(a, b) = w_{ij}(b, a) = 0$). Dans l'ordre lexicographique, $(a, b, b, a, b, b, a, b, b, a, b)$ représente une solution optimale avec un coût de optimal de 5. Une décomposition arborescente enracinée en C_1 formée des clusters $C_1 = \{1, 2, 3, 4\}, C_2 = \{4, 5, 6\}, C_3 = \{5, 6, 7\}, C_4 = \{4, 8, 9, 10\}$, et $C_5 = \{4, 9, 10, 11\}$ est présentée sur le côté droit de la figure 1. C_1 a pour fils $\{C_2, C_4\}$, le séparateur de C_3 avec son père C_2 est $S_3 = \{5, 6\}$, l'ensemble des variables propres de C_3 est $V_3 = \{7\}$. Notez que la variable 4 est une variable propre de C_1 et appartient aux séparateurs S_2, S_4 et S_5. Le sous-problème P_3 a pour variables $\{5, 6, 7\}$ et comme fonctions de coûts $\{w_{5,7}, w_{6,7}, w_7\}$. P_1 est le problème complet.*

Un algorithme de recherche peut exploiter la propriété précédente pour autant qu'un ordre des variables adapté soit utilisé : les variables d'un cluster C_e quelconque doivent être instanciées avant les variables restantes dans ses clusters fils. Dans ce cas, pour tout cluster $C_f \in \mathit{Fils}(C_e)$, une fois le séparateur S_f affecté, le sous-problème P_f conditionné par l'affectation courante A_f de S_f (notée P_f/A_f) peut être résolue à l'optimalité indépendemment du reste du problème. Le *coût optimal* de P_f/A_f peut alors être enregistré avec pour conséquence qu'il ne sera jamais plus résolu pour cette affectation du séparateur S_f. Ainsi, des algorithmes tels que BTD ou AND-OR graph search sont capables d'avoir une complexité temporelle exponentielle seulement dans la taille du plus grand cluster.

Dans [9], l'algorithme BTD exploite de minorants produits par le maintien d'une propriété de cohérence locale pondérée. Pour tout cluster C_e, une fonction de coût constante (d'arité nulle) spécifique au cluster (notée $w^e_\varnothing$) est utilisée. Elle peut être incrémentée par le mécanisme de filtrage par cohérence locale et ainsi fournir pour chaque cluster un minorant du coût d'une solution optimale. Au delà d'un meilleur élagage, ceci permet également d'éviter de résoudre systématiquement les sous-problèmes à l'optimalité en transmettant une exigence de coût maximum (un majorant) pour chaque sous-problème P_f. Pour P_1, au départ, le coût de la meilleure solution connue définit le majorant initial cub_1 . Il faut ensuite améliorer ce coût. Si l'on considère un cluster arbitraire C_f, fils de C_e avec un majorant associé cub_e, le majorant du sous-problème P_f est obtenu est *en soustrayant* de cub_e les minorants associés à tous les autres clusters fils de C_e. Ainsi, des minorants plus forts sur les autres sous-problèmes induit un majorant plus fort pour le problème courant.

Du fait de ce majorant initial, l'optimum de P_f n'est pas nécessairement calculé mais un minorant est toujours obtenu. Ce minorant et son éventuelle optimalité sont enregistrés dans LB_{P_f/A_f} et Opt_{P_f/A_f} . Ils sont initialement fixés respectivement à 0 et *faux*.

Pour un élagage amélioré et de meilleurs majorants, BTD utilise le maximum du minorant $w^e_\varnothing$ (fourni par la cohérence locale) et du minorant enregistré (LB_{P_f/A_f}). Les minorants enregistrés ne peuvent être utilisés que lorsque le séparateur S_f

est totalement instancié par l'affectation courante A. Ceci permet récursivement de définir le minorant utilisé dans [9] comme $lb(P_e/A) = w^e_\varnothing \oplus \bigoplus_{C_f \in Fils(C_e)} \max(lb(P_f/A), LB_{P_f/A_f})$.

Exemple 2 *Dans l'exemple 1, les variables* $\{1,2,3,4\}$ *de* C_1 *sont affectées en premier, en utilisant par exemple l'heuristique d'ordonnancement dynamique* min domain/max degree. *Soit* $A = \{(4,a),(1,a),(2,b),(3,b)\}$ *l'affectation courante. Le filtrage par la cohérence locale EDAC [10] sur* P_1/A *produit* $w^1_\varnothing = 2, w^2_\varnothing = w^4_\varnothing = 1, w^3_\varnothing = w^5_\varnothing = 0$, *induisant* $lb(P_1/A) = \bigoplus_{C_e \in C} w^e_\varnothing = 4$ *étant donné qu'aucun minorant n'a encore été enregistré.*

Ensuite, le sous-problème $P_2/\{(4,a)\}$ *et* $P_4/\{(4,a)\}$ *sont résolus de façon indépendante et les solutions optimales correspondantes sont mémorisées dans* $LB_{P_2/\{(4,a)\}} = 1$, $LB_{P_4/\{(4,a)\}} = 2$, $Opt_{P_2/\{(4,a)\}} = Opt_{P_4/\{(4,a)\}} = vrai$ *(aucun solution n'étant connue, aucun majorant n'est défini). Une première affectation complète de coût* $w^1_\varnothing \oplus LB_{P_2/\{(4,a)\}} \oplus LB_{P_4/\{(4,a)\}} = 5$ *est obtenue.*

L'algorithme BTD ainsi défini a une complexité temporelle qui, dans le pire des cas, est exponentielle dans la taille du plus grand cluster moins un, un nombre aussi appelé largeur d'arbre de la décomposition arborescente (C,T). Sa complexité spatiale est, dans le pire des cas, exponentielle en s où $s = \max_{C_e \in C} |S_e|$, taille du plus grand séparateur [9].

4 Poupées russes et décomposition arborescentes

L'algorithme original des poupées russes (RDS) [23], a été introduit en 1996, alors qu'aucune cohérence locale pour les CSP pondérés n'avait été définie. Son mécanisme de base a été élaboré pour construire des minorants forts à partir de minorants plus faibles en utilisant une recherche arborescente. L'application de RDS consiste à résoudre n sous-problèmes imbriqués d'un problème initial P avec n variables. Étant donné un ordre des variables statique, RDS commence par résoudre le sous-problème défini par la dernière variable seule. Ensuite, il inclut la variable précédente dans l'ordre et résout le sous-problème avec deux variables. Ce processus est répété jusqu'à la résolution compléte du problème. Chacun des sous-problèmes est résolu avec un algorithme de type DFBB utilisant un ordre statique des variables suivant l'ordre de la décomposition en sous-problèmes imbriqués. Le minorant amélioré utilisé dans l'algorithme DFBB est obtenu en combinant le minorant faible fourni par "Partial Forward Checking" (similaire au filtrage par cohérence de nœud [15]) avec l'optimum du sous-problème résolu à l'étape précédente par RDS.

Nous nous proposons d'exploiter le principe de RDS en s'appuyant sur une décomposition arborescente (RDS-BTD). La principale différence avec RDS réside dans le fait que l'ensemble des sous-problèmes résolus est défini par une décomposition arborescente enracinée (C,T) : RDS-BTD résout $|C|$ sous-problème ordonnés par un parcours en profondeur de T, démarrant aux feuilles jusqu'à la racine $P^{\text{RDS}}_1 = P_1$.

Nous définissons P^{RDS}_e comme le sous-problème défini par les variables *propres* de C_e ainsi que par celles de tous ses clusters descendants dans T et par les fonctions de coût impliquant *uniquement* des variables propres de ces clusters. P^{RDS}_e n'a aucun fonction de coût impliquant une variable de S_e, le séparateur avec son cluster père, et ainsi son coût optimal est un minorant de P_e conditionné par *n'importe quel* affectation de S_e. Ce coût optimal sera donc noté $LB^{\text{RDS}}_{P_e}$

Chaque sous-problème P^{RDS}_e est résolu par BTD et non par DFBB. Ceci permet d'exploiter la décomposition et la mémorisation de minorants effectuée dans BTD, offrant ainsi une complexité asymptotique améliorée. Le minorant exploité peut alors s'appuyer sur les minorants $LB^{\text{RDS}}_{P_e}$ déjà calculés sur des clusters déjà explorés, sur les minorants fournis par la cohérence locale et sur les minorants enregistrés par BTD[1]. Le minorant correspondant à l'affectation courante A est maintenant défini de façon récursive par $lb(P_e/A) = w^e_\varnothing \oplus \bigoplus_{C_f \in Fils(C_e)} \max(lb(P_f/A), LB_{P_f/A_f}, LB^{\text{RDS}}_{P_f})$, qui est évidemment plus fort.

Dans BTD, la mémorisation n'est effectuée que lorsque les séparateurs sont complètement affectés alors que P^{RDS}_e ne contient pas les variables du séparateur S_e. Nous affectons donc S_e avant de résoudre P^{RDS}_e en utilisant les valeurs totalement supportées (full support) fournies par EDAC [10][2] de chaque variable comme des valeurs temporaires utilisées uniquement pour la mémorisation. Une approche alternative consisterait à mémoriser les minorants d'affectations partielles mais ceci nécessiterait un mécanisme de mémorisation complexe, à considérer dans des travaux futurs.

L'avantage de l'utilisation de BTD réside aussi dans le fait que les minorants enregistrés peuvent aussi être exploités dans les itérations suivantes de RDS-

[1] En fait, du fait que le filtrage par cohérence locale peut déplacer des coûts entre les clusters, le minorant $LB^{\text{RDS}}_{P_f}$ doit être ajusté. Ceci est rendu possible par l'utilisation de la structure de données ΔW introduite dans [9] en soustrayant $\bigoplus_{i \in S_f} \max_{a \in D_i} \Delta W^f_i(a))$.

[2] Une valeur $a \in D_i$ totalement supportée vérifie $w_i(a) = 0$ et $\forall w_S \in W$ *with* $i \in S, \exists t \in \ell(S)$ *with* $t[i] = a$ tel que $w_S(t) = 0$.

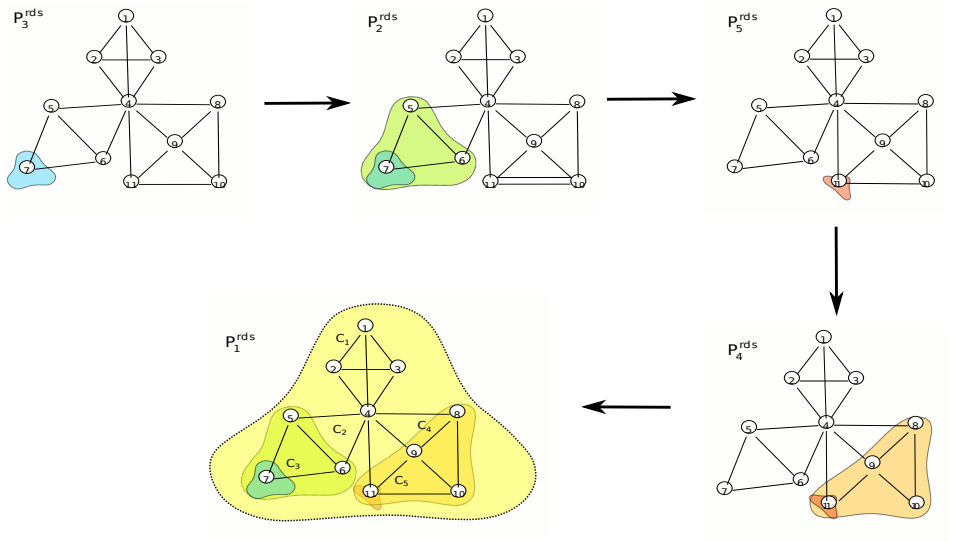


FIG. 2 – Résolution récursive des sous-problèmes relaxés issus de la Figure 1 par RDS-BTD.

BTD. Cependant, un optimum mémorisé par BTD pour un sous-problème P_f alors que P_e^{RDS} était résolu n'est plus nécessairement un optimum de $P_{Père(e)}^{\text{RDS}}$ si une fonction de coût existe entre P_f et les variables de $C_{Père(e)}$. De ce fait, à chaque itération de RDS-BTD, après la résolution de P_e^{RDS}, tous les drapeaux Opt_{P_f/A_f} tels que $S_f \cap S_e \neq \varnothing$ sont réinitialisés (line 4).

Exemple 3 *Considérons l'exemple 1, RDS-BTD traite successivement les sous-problèmes* P_3^{RDS}, P_2^{RDS}, P_5^{RDS}, P_4^{RDS} *et* $P_1^{\text{RDS}} = P_1$. P_3^{RDS} *contient seulement la variable* $\{7\}$ *et la fonction de coût* $\{w_7\}$. *Avant de résoudre* P_3^{RDS}, *RDS-BTD affecte les variables* $\{5, 6\}$ *du séparateur* S_3 *à leur valeurs totalement supportées* ($\{(5, a), (6, a)\}$ *ici). Quand* P_2^{RDS} *est résolu, l'optimum de* $P_3/\{(5, a), (6, a)\}$, *qui est égal à zéro car* $w_{5,6}$ *n'appartient pas à* P_3, *est enregistré et peut être réutilisé lors de la résolution de* P_1. *Quand* P_4^{RDS} *est résolu, l'optimum de* $P_5/\{(4, a), (9, a), (10, a)\}$, *qui est également nul, est enregistré. Dans ce cas, du fait que la variable* 4 *appartient à* $S_5 \cap S_4$ *et puisque* P_4^{RDS} *ne contient pas* $w_{4,11}$, *cet optimum enregistré n'est en fait qu'un minorant pour les itérations suivantes de RDS-BTD. De ce fait, nous fixons les booléens* $Opt_{P_5/\{(4,a),(9,a),(10,a)\}}$ *à faux avant de résoudre* P_1.

Les optima obtenus sont $LB_{P_3}^{\text{RDS}} = LB_{P_5}^{\text{RDS}} = 0$, $LB_{P_2}^{\text{RDS}} = LB_{P_4}^{\text{RDS}} = 1$ *et* $LB_{P_1}^{\text{RDS}} = 5$, *l'optimum de* P_1.

Dans cet exemple simple, pour une affectation initiale vide $A = \varnothing$, *on obtient un minorant* $lb(P_1/\varnothing) = LB_{P_2^{\text{RDS}}} \oplus LB_{P_4^{\text{RDS}}} = 2$ *au lieu de* 0 *pour BTD (en faisant l'hypothèse que la cohérence locale EDAC est utilisée en pré-traitement et qu'aucun majorant initial n'est fourni).*

Le pseudo-code de l'algorithme RDS-BTD est présenté ci-aprés comme l'algorithme 1. L'algorithme BTD utilisé est le même que dans [9] si ce n'est pour le fait qu'il exploite le minorant RDS amélioré défini ci-dessus. Nous faisons l'hypothèse que la résolution de P_e^{RDS} avec une affectation initiale A du séparateur S_e et un majorant initial cub_e obtenu par BTD est efféctuéé par l'appel à $\mathsf{BTD}(P_e^{\text{RDS}}, A, V_e, cub_e)$.

RDS-BTD appelle BTD pour résoudre chaque sous-problème P_e^{RDS} (ligne 3). Un majorant initial pour P_e^{RDS} est déduit du majorant global et des minorants RDS disponibles (ligne 1). Comme expliqué ci-dessus, les variables de S_e sont affectées à leurs valeurs totalement supportées à la ligne 2 . L'appel initial est simplement $\mathsf{RDS\text{-}BTD}(P)$. Il suppose que le problème $P_1^{\text{RDS}} = P$ est déjà localement cohérent et retourne son coût optimal.

Notez que dès qu'une solution de P_e^{RDS} ayant le même coût optimal que $lb(P_e^{\text{RDS}}/\varnothing) = \bigoplus_{C_f \in Fils(C_e)} LB_{P_f^{\text{RDS}}}$ est trouvée, la recherche s"arrête.

Les complexités temporelles et spatiales de RDS-BTD sont celles de BTD. Notez que sans mémorisation, en utilisant uniquement un filtrage par cohérence de nœud et une décomposition induite par un pseudo-arbre (avec un cluster pour chaque variable et un ordre statique des variables), RDS-BTD est équivalent à l'algorithme Pseudo-Tree RDS [14]. Si

Algorithm 1: RDS-BTD algorithme

Function RDS-BTD(P_e^{RDS}) : $[0, +\infty]$
  foreach $C_f \in Sons(C_e)$ **do** RDS-BTD(P_f^{RDS}) ;
1 $cub_e := cub_1 - lb(P/\varnothing) + lb(P_e^{\text{RDS}}/\varnothing)$;
2 Soit A l'affectation de S_e à des valeurs totalement supportées;
3 $LB_{P_e}^{\text{RDS}} :=$ BTD(P_e^{RDS}, A, V_e, cub_e) ;
  foreach C_f descendant de C_e *t.q.* $S_f \cap S_e \neq \varnothing$ **do**
4 Fixer à *faux* les $Opt_{P_f/A}$ enregistrés, pour tout $A \in \ell(S_f)$;
  return $LB_{P_e}^{\text{RDS}}$;

de plus, nous restreignons l'algorithme à l'utilisation d'une décomposition arborescente spécifique (C, T) telle que $|C| = n, \forall e \in [1, n], C_e = \{1, \ldots, e\}$, et $\forall e \in [2, n], Père(C_e) = C_{e-1}$, alors BTD-RDS se réduit à RDS.

5 Résultats expérimentaux

Nous avons codé DFBB, BTD, RDS-BTD, et RDS dans `toulbar2` un solveur C++ de CSP pondérés[3]. A l'intérieur des clusters, l'heuristique dynamique de choix de variable *min domain / max degree* est utilisée (par BTD et RDS-BTD) et par DFBB, les ex-aequo sont classés par les coûts unaires maximum. Cette heuristique est modifiée par une heuristique de type "conflict back-jumping" comme cela est proposé dans [16]. Durant la recherche, un niveau de cohérence locale EDAC [10] est maintenu à chaque étape. RDS s'appuie sur la cohérence de nœud [15] seulement. Les décompositions arborescentes sont construites par l'heuristique "Maximum Cardinality Search" (MCS), en utilisant le plus grand cluster comme racine (excepté pour le problème `scen07`). À partir de la décomposition arborescente fournie par MCS, nous avons dérivé des décompositions arborescentes alternatives définies par une taille de séparateur maximum s_{max} comme cela est proposé dans [13] : en partant des feuilles de la décomposition, nous fusionnons un cluster avec son père si la taille du séparateur dépasse s_{max}. Un ordre des variables compatible avec la décomposition arborescente enracinée exploitée est utilisé pour l'établissement de la cohérence locale DAC [6] ainsi que par RDS.

Les minorants enregistrés (et si disponibles, fournis par RDS) sont exploités par le filtrage par cohérence locale dés que le séparateur correspondant est totalement affecté. Si le minorant enregistré est optimal ou supérieur au minorant fourni par EDAC, le sous-problème (P_e/A_e) correspondant est déconnecté du filtrage par cohérence locale et la différence positive entre les minorants est ajoutée au minorant du cluster père ($w_{\varnothing}^{Père(C_e)}$), conduisant à de nouveaux effacements de valeurs par cohérence de nœud.

Toutes les méthodes s'appuient sur un schéma de branchement binaire. Si $d > 10$, le domaine *ordonné* est coupé en deux parties (autour de la valeur centrale) sinon, la variable est affectée à son support complet ou cette valeur est retirée du domaine. Dans les deux cas, la branche contenant la valeur support complet est explorée en premier, excepté pour RDS et les méthodes dérivées de BTD qui choisissent en priorité (si elle existe) la branche qui contient la valeur de la dernière solution trouvée. Sauf mention précise, aucun majorant initial n'est fourni. Les temps CPU présentés correspondent au temps nécessaire pour trouver une solution optimale et prouver son optimalité.

5.1 Affectation de fréquences

Parmi les différentes instances du CELAR [2] qui peuvent être décrites comme un CSP pondéré binaire, nous avons sélectionné deux instances difficiles : `scen06` and `scen07`. L'instance `scen07` a été réduite par différentes règles de prétraitement (filtrage des domaines par singleton EDAC, par les règles de dominance proposées par Koster [8], élimination des variables de degré faible) conduisant à une instance où $n = 162$, $d = 44$ et $e = 764$. La décomposition arborescente utilisée a un $s_{max} = 3$ et une largeur d'arbre de 53. Un majorant initial (354008) a été fourni à toutes les méthodes. Ce majorant et la solution correspondante ont été trouvés par un algorithme de type DFBB utilisant un mécanisme de type "Limited Discrepancy Search". Cette solution est également utilisée par RDS-BTD comme choix de valeur initial (les performances de DFBB et BTD sont dégradées par ce choix). Le cluster racine a été choisi sur la base du "First Fail Principle" (cluster de coût optimal maximum). BTD et RDS-BTD trouvent l'optimum de 343592 et prouvent l'optimalité en respectivement 6 et 4.5 jours (amélioration de 26% pour RDS-BTD) sur un processeur à 2.6 GHz équipé de 32GB de mémoire. BTD a enregistré 90528 minorants (20% de l'espace des affectations de séparateurs) comparés aux 29453 (6.4%) mémorisés par RDS-BTD. DFBB n'a pas terminé en 50 jours. C'est la première résolution avec preuve d'optimalité de ce problème ouvert depuis plus de 10 ans.

Nous avons aussi résolu l'instance `scen06` (100 variables avec une largeur d'arbre de 11) sans aucun prétraitement ni majorant initial. BTD et RDS-BTD ont pris 221 et 316 secondes respectivement pour trouver l'optimum et prouver l'optimalité. Ils ont respective-

[3] https ://mulcyber.toulouse.inra.fr/projects/toulbar2 version 0.8.

ment enregistré 5633 et 7145 minorants malgré le fait que la taille des séparateurs n'ait pas été restreinte (on observe un $s_{max} = 8$). DFBB a pris 2588 secondes et RDS n'a pas terminé en 10 heures.

5.2 Sélection de Tag SNP

Ce problème apparaît en génétique et analyse du polymorphisme. Les SNP (ou Single Nucleotide Polymorphism, prononcé snip) sont des variations ponctuelles dans le génome d'individus d'une même espèce. Il se caractérise par une altération d'un seul nucléotide (`A`,`T`,`C`,or `G`) dans la séquence. Par exemple, un SNP peut changer la séquence D'ADN `AAGGCTAA` en `ATGGCTAA`. Pour qu'une variation soit considérée comme un SNP, elle doit apparaître dans au moins 1% de la population considérée. Dans les trois milliards de nucléotides que compte le génome humain, on dénombre plusieurs millions de SNP. Ils expliquent jusqu'à 90% de toutes les variations génétiques humaines, notamment une partie du risque héritable de maladies communes ainsi que la susceptibilité de réponses aux pathogènes, produits chimiques, médicaments vaccins et autres agents.

Le problème TagSNP est une forme de compression d'information avec perte consistant à sélectionner un sous-ensemble de SNP tel que les SNP sélectionnés (appelés tag SNP) capturent l'essentiel de l'information génétique. Le but est de capturer un ensemble de petite taille le plus informatif possible afin de rendre possible le criblage et l'analyse statistique d'une population de grande taille [12].

La mesure de corrélation r^2 entre une paire de SNP peut dans un premier temps être déterminée sur une petite population. Un tag SNP est considéré comme "représentatif" d'un autre SNP si les deux SNP sont suffisamment corrélés. Le problème de sélection de tag SNP le plus simple consiste à sélectionner un nombre minimum de tag SNP de façon à ce que tous les SNP soient représentés. Ceci est capturé par le fait que la mesure r^2 entre deux SNP est supérieure à un seuil θ (souvent fixé à $\theta = 0.8$ [4]). Nous considérons donc un graphe dans lequel chaque sommet est un SNP et où les arêtes sont pondérées par la mesure r^2 entre les SNP des sommets. Les arêtes dont le seuil est plus petit que θ sont supprimées. Le graphe filtré ainsi obtenu peut avoir plusieurs composantes connexes. Le problème TagSNP se réduit alors à un problème de couverture d'ensemble (set covering, NP-dur) sur chacune des composantes. Cette formulation simple du problème a été étudiée avec de bons résultats dans [1] via le compilateur `c2d` .

En pratique, le nombre de solutions optimales peut être très important et les outils spécialisés du domaines tel que FESTA [18], (en s'appuyant sur deux algorithmes incomplets), optimisent, en plus du nombre de tagsnp selection, des critères secondaires : Entre deux tag SNP, une mesure r^2 faible est préférée afin de maximiser la dispersion des tag SNP. Entre un non tag SNP et un de ses représentants, une mesure r^2 élevée est préférée afin de maximiser la représentativité de la sélection de tag SNP effectuée.

Pour un graphe connexe donné $G = (V, E)$, nous construisons un WCSP binaire avec des coûts entiers capturant le problème TagSNP avec les critères secondaires précédents. Pour chaque SNP i, deux variables i_s et i_r sont introduites. i_s est une variable booléenne indiquant si le SNP est sélectionné comme tag SNP ou non. Le domaine de i_r est formé par l'ensemble des voisins de i complété avec i lui-même. Il indique le tag SNP qui est chargé de représenter i. Pour un SNP i, des fonctions de coût dures (avec des coûts nuls ou infinis) établissent le fait que $i_s \Rightarrow (i_r = i)$. Des fonctions de coût dures similaires établissent $(i_r = j) \Rightarrow j_s$ avec les SNP j voisins dans G. Une fonction de coût unaire sur chaque variable i_s produit un coût élémentaire U si la variable est à *vrai*. Le WCSP résultant capture déjà le problème de couverture d'ensembles pur défini par le problème TagSNP.

Pour prendre en compte le critère de représentativité, une fonction de coût unaire est associée avec chaque variable i_r . Si $i_r \neq i$, elle génère un coût non nul égal à $\lfloor 100.\frac{1-r^2_{i,i_r}}{1-\theta} \rfloor$. Pour capturer la dispersion entre tag SNPs i et j, une fonction de coût binaire entre les booléens i_s et j_s est introduite. Elle produit un coût de $\lfloor 100.\frac{r^2_{ij}-\theta}{1-\theta} \rfloor$ lorsque $i_s = j_s = vrai$. Le WCSP ainsi obtenu capture simultanément les critères de dispersion et de représentativité. Afin de garder à ces critères leur caractère secondaire, nous utilisons simplement une valeur de coût U (le coût de sélection d'un SNP) suffisamment grande. En pratique, supérieur à la somme des critère secondaire et primaire dans l'instance considérée.

Ce problème est similaire à un problème de couverture d'ensembles (set covering) mais avec des coûts additionnels binaires (quadratiques). Ces critères sont ignorés par [1]. Dans cet article, `c2d` donne une représentation compacte de l'ensemble des solutions du problème de couverture simple, mais le nombre de solutions optimales est si grand (typiquement supérieur à des milliards) que l'application des critères secondaires à des solutions générées par `c2d` serait trop coûteuse. Comme le mentionnent les auteurs dans leur conclusion, une compilation directe des critères secondaire dans le d-DNNF ne semble pas immédiate. Elle nécessiterait une reformulation du problème sous forme MaxSAT.

Les instances considérées dérivent de données obtenues sur chromosome humain numéro 1, aimable-

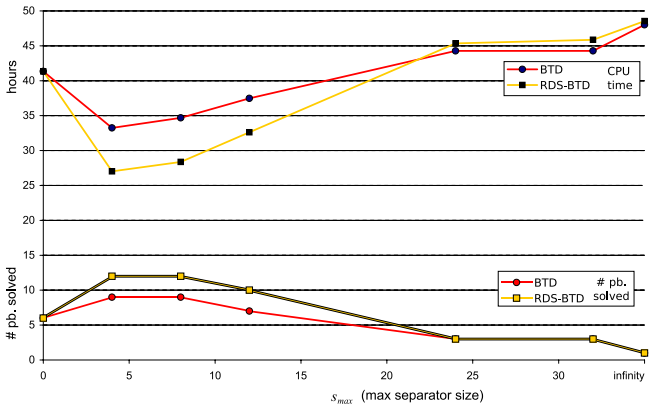


FIG. 3 – Temps de résolution global et nombre de problèmes résolus en faisant varier la taille maximale du séparateur s_{max}.

ment fournies par Steve Qin [18]. Deux valeurs de seuil, $\theta = 0.8$ et 0.5 ont été essayées. Pour $\theta = 0.8$, une valeur usuelle dans le domaine, 43251 composantes connexes ont été identifiées parmi lesquelles nous avons sélectionné les 82 plus grandes. Ces problèmes, qui incluent de 33 à 464 SNP, définissent des WCSP avec des tailles de domaine entre 15 et 224. Ces instances sont relativement faciles. Leur résolution à l'optimalité sélectionne 359 tag SNPs en 2h37' au lieu des 487 SNP sélectionnés en 3' par FESTA (en mode greedy), soit 21% d'amélioration. En mode hybride, FESTA sélectionne 370 tag SNP en 39h17' soit un modeste gain de 3% dans notre cas, mais avec une amélioration d'un facteur 15 en temps.

Afin d'obtenir des problèmes plus difficiles, nous avons abaissé le seuil θ à 0.5. Ceci défini $19,750$ composantes connexes parmi lesquelles 516non triviales sont résolues par FESTA par approches incomplètes. Nous avons sélectionnées les 25 plus grandes. Ces instances contiennent de 171 à 777 SNP et ont une densité d'arête qui varie entre 6% et 37%. Elles définissent des instances WCSP dont la taille des domaines varie entre 30 et 266 et contenant de 8000 à $250,000$ fonctions de coûts. La décomposabilité de ces problèmes, estimée par le ratio entre la largeur d'arbre d'une décomposition fournie par MCS ($s_{max} = +\infty$) et le nombre de variables, varie de 14% à 23%.

Tous les problèmes ont été traités avec un majorant initial fourmi par FESTA (mode greedy). L'éxperimentation a été réalisé sur machine équipée de CPU à 2.8 Ghz et 32 GB de RAM. Pour mettre en valeur l'importance de l'utilisation d'une taille de séparateurs bornée (via s_{max}), nous avons considéré des valeurs de s_{max} allant de 0 (DFBB), 4, 8, 12, 24, 32 à $+\infty$ pour les algorithmes BTD et RDS-BTD. Nous présentons le nombre de problèmes résolus dans une limite de temps de deux heures ainsi que le temps CPU global utilisé (une instance non résolue comptant pour deux heures). RDS présentant de mauvaises performances, les résultats correspondants ne sont pas présentés.

En utilisant $s_{max} = 4$, notre implémentation améliore le taux de compression de FESTA-greedy par un rapport de 15% (en sélectionnant 2952 tag SNP au lieu de 3477 pour les $516-13$ instances résolues). Notez que la différence en temps CPU entre BTD et RDS-BTD augmenterait si une plus grande limite de temps était utilisée. D'un point de vue pratique, le critère utilisé dans le problème TagSNP pourrait être raffinée encore en incluant des informations variées : informations d'annotation de séquences (*e.g.* préférant les tag SNP apparaissant dans les gènes), mesure entre triplets de SNP comme proposé dans [1] (SNP couverts par des paires de tag SNP). Les bonnes performances de RDS-BTD pourraient permettre de résoudre ces problèmes avec un seuil réaliste de $\theta = 0.8$.

La Figure 4 représente, pour chaque instance ($\theta = 0.5$) résolue au moins une fois lors des expérimentations précédentes, l'évolution individuelle du ratio largeur d'arbre de la décomposition sur taille du problème en fonction de s_{max}. Le succès de la résolution (preuve d'optimalité obtenue dans le temps imparti) est matérialisé par un trait continu, à l'inverse les échecs sont représentés par des pointillés. L'unique résolution de l'instance "17034" (s_{max}=4), est représentée par un carré gris.

Ainsi, on retrouve individuellement les 12 instances résolues pour $s_{max} = \{4, 8\}$, mais il est intéréssant de noter que la nature des instances clôturées sont légèrement différentes. L'instance "17034" est résolue uniquement à s_{max}=4, ce qui est compensé par l'instance "14359" , qui est résolue sur l'intervalle 8 et 12. Au total, 13 instances sont clôturées sur l'ensemble des

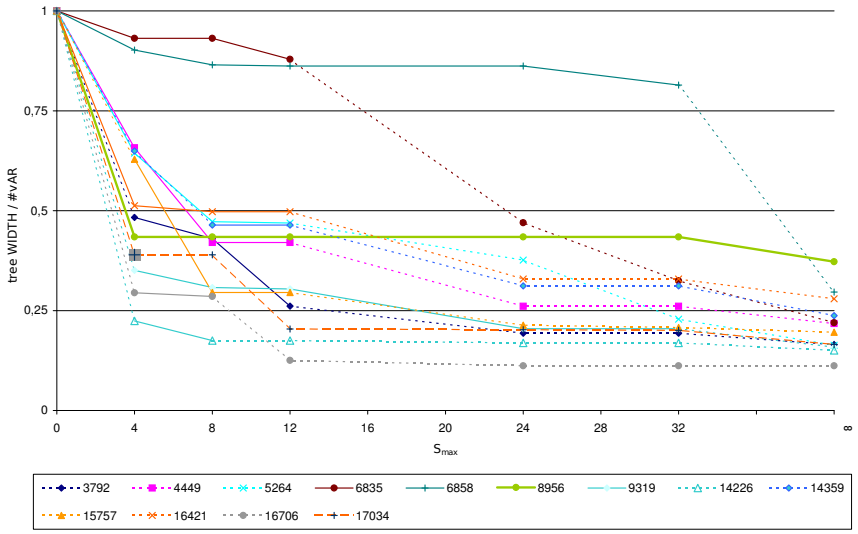


FIG. 4 – Evolution de la largeur d'arbre normalisée par le nombre de variables en fonction de la taille maximale des séparateurs s_{max}.

expérimentations. Une diminution importante dans la largeur d'arbre de décomposition est souvent corrélée au succès de la résolution du problème.

Pour la majorité des instances étudiées, reflétant la structure des problémes, les courbes montrent une zone de décroissance importante pour $s_{max} \in [0, 8]$, qui est sans doute à l'origine des bons résultats obtenus. La diminution de la largeur d'arbre est ensuite plus faible. Elle atteint progressivement ($s_{max} > 12$), un plateau de pente faible, qui tends asymptotiquement vers le ratio associé à la décomposition complète du problème. Cette zone des courbes est souvent associée à un échec dans les expérimentations. Ainsi par exemple, l'instance 15757 est résolue pour des valeurs de s_{max} de 4, 8, 12 et 24 avec respectivement des temps de 514, 7, 299 et 3810 secondes. Sur le plan de la décomposition, le ratio de largeur d'arbre passe de $0,63$ pour s_{max}=4 à $0,30$ pour $s_{max} = 8$ et $s_{max} = 12$ pour enfin avoisiner $0,2$ à $s_{max} = 24$. La résolution de l'instance est optimale à $s_{max} = 8$. Le cas de cette instance n'est pas isolé. Comme le montre la Figure 3, la plupart des instances résolues présentent une valeur s_{max} *optimale* pour la résolution. Cette situation correspond, probablement, à un bon compromis entre gains issus de la décomposition et pertes liées à une diminution dans la liberté de l'ordre de choix des variables.

6 Conclusion

L'exploitation pratique de décompositions arborescentes pour résoudre des problèmes d'optimisation combinatoire qui ont une structure n'est pas toujours immédiate. Les problèmes d'optimisation définissent de façon explicite un critère global qui doit être optimisé sur l'ensemble de l'instance. En résolvant des sous-problèmes conditionnellement indépendants, les algorithmes exploitant une décomposition arborescente des problèmes perdent la vision globale sur le critère en exploitant des bornes locales assez faibles. En fournissant des minorants forts pour chacun des sous-problèmes, l'approche des poupées russes permet d'injecter une information plus globale dans chaque résolution de sous-problème au travers d'un majorant initial renforcé, évitant ainsi nombre de résolution inutiles et menant à une efficacité accrue.

Au delà, nos expérimentations montrent que, même sur des problèmes ayant une structure forte, il est souvent profitable et parfois indispensable de restreindre la décomposition en bornant la taille des séparateurs. La théorie nous indique que la taille des séparateurs influe sur la complexité spatiale d'algorithmes tels que BTD ou RDS-BTD, mais même lorsque la taille des séparateurs n'est pas restreinte, aucun des instances

considérées dans nos tests n'a pu épuiser la mémoire disponible. En pratique, l'amélioration en efficacité est d'abord explicable par la liberté supplémentaire dans l'ordonnancement des variables rendue possible par la fusion de clusters. Cette observation est cohérente avec les conclusions de [13]. On notera aussi que le nombre d'instanciations possibles d'un séparateur croît exponentiellement avec sa taille. Plus le séparateur est grand, et plus la probabilité qu'un minorant mémorisé serve à nouveau devient faible : la mémorisation est sans doute surtout utile dans les séparateurs de petite taille.

Concernant notre algorithme, la synergie entre RDS et BTD pourrait être améliorée en autorisant BTD à mémoriser des minorants associés à des instanciations partielles des séparateurs. Cela permettrait de réduire encore les recherches redondantes entre itérations successives de RDS-BTD et serait également profitable dans une stratégie d'approfondissement itératif (iterative deepening) comme cela a été proposé dans [3].

Dans nos tests, le seuil s_{max} a été appliqué de façon uniforme à toutes les instances. Il reste à étudier à quel point un ajustement de ce seuil par instance, en prenant en compte la taille des séparateurs existants dans le problème et les variations de largeur d'arbre qu'ils peuvent induire, pourrait améliorer les performances de l'algorithme.

Acknowledgments Ce travail a été partiellement financé par l'*Agence Nationale de la Recherche* (projet `STALDECOPT`).

Références

[1] A. Choi, N. Zaitlen, B. Han, K. Pipatsrisawat, A. Darwiche, and E. Eskin. Efficient Genome Wide Tagging by Reduction to SAT. In *Proc. of WABI-08*, volume 5251 of *LNCS*, pages 135–147, 2008.

[2] B. Cabon, S. de Givry, L. Lobjois, T. Schiex, and J.P. Warners. Radio link frequency assignment. *Constraints Journal*, 4 :79–89, 1999.

[3] B. Cabon, S. de Givry, and G. Verfaillie. Anytime Lower Bounds for Constraint Optimization Problems. In *Proc. of CP-98*, pages 117–131, Pisa, Italy, 1998.

[4] C. S. Carlson, M. A. Eberle, M. J. Rieder, Q. Yi, L. Kruglyak, and D. A. Nickerson. Selecting a maximally informative set of single-nucleotide polymorphisms for association analyses using linkage disequilibrium. *Am. J. Hum. Genet.*, 74(1) :106–120, 2004.

[5] M. Cooper, S. de Givry, and T. Schiex. Optimal soft arc consistency. In *Proc. of IJCAI-07*, pages 68–73, Hyderabad, India, 2007.

[6] M. Cooper and T. Schiex. Arc consistency for soft constraints. *Artificial Intelligence*, 154 :199–227, 2004.

[7] A. Darwiche. Recursive Conditioning. *Artificial Intelligence*, 126(1-2) :5–41, 2001.

[8] S. de Givry. Singleton consistency and dominance for weighted CSP. In *Proc. of Soft Constraints workshop*, 2004.

[9] S. de Givry, T. Schiex, and G. Verfaillie. Exploiting Tree Decomposition and Soft Local Consistency in Weighted CSP. In *Proc. of AAAI-06*, Boston, MA, 2006.

[10] S. de Givry, M. Zytnicki, F. Heras, and J. Larrosa. Existential arc consistency : Getting closer to full arc consistency in weighted CSPs. In *Proc. of IJCAI-05*, pages 84–89, Edinburgh, Scotland, 2005.

[11] E. Freuder and M. Quinn. Taking advantage of stable sets of variables in constraint satisfaction problems. In *Proc. of IJCAI-85*, pages 1076–1078, Los Angeles, CA, 1985.

[12] J.N. Hirschhorn and M.J. Daly. Genome-wide association studies for common diseases and complex traits. *Nature Reviews Genetics*, 6(2) :95–108, 2005.

[13] P. Jégou, S. N. Ndiaye, and C. Terrioux. Dynamic management of heuristics for solving structured CSPs. In *Proc. of CP-07*, pages 364–378, Providence, USA, 2007.

[14] J. Larrosa, P. Meseguer, and M. Sanchez. Pseudo-tree search with soft constraints. In *Proc. of ECAI-02*, pages 131–135, Lyon, France, 2002.

[15] J. Larrosa and T. Schiex. Solving Weighted CSP by Maintaining Arc-consistency. *Artificial Intelligence*, 159(1-2) :1–26, 2004.

[16] C. Lecoutre, L. Sais, S. Tabary, and V. Vidal. Last conflict based reasoning. In *Proc. of ECAI-2006*, pages 133–137, Trento, Italy, 2006.

[17] R. Marinescu and R. Dechter. Memory intensive branch-and-bound search for graphical models. In *Proc. of AAAI-06*, Boston, MA, 2006.

[18] Z. S. Qin, S. Gopalakrishnan, and G. R. Abecasis. An efficient comprehensive search algorithm for tagsnp selection using linkage disequilibrium criteria. *Bioinformatics*, 22(2) :220–225, 2006.

[19] M. Sanchez, S. de Givry, and T. Schiex. Mendelian error detection in complex pedigrees using weighted constraint satisfaction techniques. *Constraints*, 13(1) :130–154, 2008.

[20] T. Schiex. Arc consistency for soft constraints. In *Proc. of CP-2000*, pages 411–424, Singapore, 2000.

[21] T. Schiex, H. Fargier, and G. Verfaillie. Valued constraint satisfaction problems : hard and easy problems. In *Proc. of the 14^{th} IJCAI*, pages 631–637, Montréal, Canada, 1995.

[22] C. Terrioux and P. Jegou. Hybrid backtracking bounded by tree-decomposition of constraint networks. *Artificial Intelligence*, 146(1) :43–75, 2003.

[23] G. Verfaillie, M. Lemaître, and T. Schiex. Russian Doll Search for Solving Constraint Optimization Problems. In *Proc. of AAAI-96*, pages 181–187, Portland, OR, 1996.

Comptage de solutions en exploitant la structure du graphe de contraintes

Aurélie Favier, Simon de Givry et Philippe Jégou

INRA MIA Toulouse, France
Université Paul Cézanne, Marseille, France
{afavier,degivry}@toulouse.inra.fr, philippe.jegou@univ-cezanne.fr

Abstract

Ce papier traite du problème de comptage du nombre de solutions d'un CSP, dénoté #CSP. Ce problème très difficile a de nombreuses applications en informatique, en particulier en Intelligence Artificielle, et aussi en physique statistique. Des progrès récents ont été fait par des méthodes de recherche, telle que BTD [16], qui exploitent la structure du graphe de contraintes dans le but de résoudre des CSPs efficacement. Nous proposons d'adapter BTD pour résoudre le problème #CSP. La méthode de comptage exacte résultante a dans le pire des cas une complexité temporelle exponentielle en un paramètre graphique spécifique appelé *largeur d'arbre*. Pour des problèmes ayant un graphe peu dense mais une grande largeur d'arbre, nous proposons une méthode itérative qui approche le nombre de solutions en résolvant une partition de l'ensemble des contraintes en une collection de sous-graphes partiels triangulés. Sa complexité temporelle est exponentielle en la taille de la plus grande clique (aussi appelé *clique number*) du problème d'origine, taille qui peut être beaucoup plus petite que la largeur d'arbre. Des expérimentations sur des problèmes aléatoires CSP structurés et des benchmarks SAT montrent l'efficacité pratique de nos approches.

1 Introduction

Le formalisme des problèmes de satisfaction de contraintes (CSP) offre un cadre général pour représenter et résoudre de nombreux problèmes. Déterminer si une solution existe est un problème NP-complet. Un problème plus difficile encore consiste à compter le nombre de solutions. Ce problème noté #CSP est connu pour être #P-complet [26]. Ce problème a de nombreuses applications en informatique, en particulier en IA, par exemple en raisonnement approché [23], en diagnostique [19], en révision de croyance [5], comme heuristique pour guider la recherche d'une solution dans les CSPs [17], ainsi que dans d'autres domaines en dehors de l'informatique, tels que la physique statistique [3] ou en biochimie pour la prédiction de structures de protéines [20].

Dans la littérature, deux principales approches ont été étudiées. D'un côté, les méthodes calculant le nombre exact de solutions et de l'autre côté des méthodes approchées. Pour les méthodes exactes, l'approche naturelle est d'étendre les méthodes systématiques telles que FC [15] ou MAC [24] dans le but d'énumérer toutes les solutions. La complexité est bornée par $O(m.d^n)$ avec n le nombre de variables, m le nombre de contraintes, et d la taille maximum des domaines. Il est évident qu'avec cette approche, plus il y a de solutions,

plus cela prend du temps pour les énumérer.

Dans ce papier, nous sommes intéressés par des méthodes de recherche arborescente qui exploitent la structure des problèmes, offrant de meilleures bornes sur la complexité temporelle et spatiale. C'est le cas du compilateur de formules en logique propositionnelle en d-DNNF [6] et de la recherche dans des graphes ET/OU [8, 9] qui font tous les deux du comptage de solutions.

Nous proposons d'adapter l'algorithme Backtracking with Tree-Decomposition (BTD) [16] à #CSP. BTD avait été initialement proposé pour résoudre des CSPs structurés. Notre modification sur BTD est similaire à celle effectuée dans le cadre de la recherche ET/OU [8, 9]. Cependant BTD utilise la notion d'arbre de décomposition de clusters au lieu d'un pseudo-arbre, ce qui conduit naturellement BTD à utiliser un ordre dynamique de choix de variables à l'intérieur des clusters, tandis que la recherche ET/OU utilise plutôt un ordre statique.

La plus part du travail réalisé sur le comptage a été fait sur #SAT, le problème de comptage de modèles associé à SAT [26]. Les méthodes exactes pour #SAT étendent les résolveurs SAT systématiques, en ajoutant une analyse des composants [2] et de la mémorisation [25] pour améliorer les performances.

Les approches qui réalisent une approximation proposent une estimation du nombre de solutions. Elles proposent des algorithmes en temps polynômial ou exponentiel qui doivent fournir des approximations de qualité raisonnable avec des garanties théoriques sur la qualité de l'approximation ou non. De même que pour les méthodes exactes, les principaux travaux sur les méthodes d'approximation ont été fait sur #SAT. Ces approches effectuent soit un échantillonnage dans l'espace de recherche d'origine [27, 12, 10, 18], soit dans l'espace de recherche ET/OU [11]. Toutes ces méthodes, à l'exception de [27] procurent un minorant probabiliste du nombre de solutions avec un intervalle de confiance très resserré obtenu par des affectations aléatoires des variables jusqu'à trouver des solutions. Un inconvénient possible de ces approches est qu'elles peuvent ne trouver aucune solution dans le temps de calcul imparti à cause du fait qu'elles rencontrent uniquement des affectations partielles incohérentes. Pour des problèmes complexes de grande taille, cela conduit à des minorants du nombre de solutions nuls. Pour tenter de remédier à ce problème, il faut alors réaliser un réglage délicat des paramètres de ces méthodes, par exemple en changeant le nombre d'échantillons. Une autre approche consiste à réduire l'espace de recherche en ajoutant par exemple des contraintes XOR [13, 14]. Cependant, l'ajout de ces contraintes n'assure pas que le problème soit plus facile à résoudre.

Dans ce papier, nous proposons de relaxer le problème de comptage en effectuant un partitionnement des contraintes en une collection de sous-problèmes structurés triangulés. Chaque sous-problème est alors résolu en utilisant notre version modifiée de BTD. Cette tâche devrait être relativement facile si l'instance originale a un graphe peu dense[1]. Finalement, une approximation du nombre de solutions du problème complet est obtenue en combinant les résultats obtenus pour chaque sous-problème. La méthode d'approximation résultante, appelée `ApproxBTD` par la suite, donne aussi un majorant trivial du nombre de solutions. Les résultats expérimentaux montrent qu'une telle approche est intéressante pour sa rapidité et la qualité de son approximation.

D'autres méthodes de comptage fondées sur des relaxations ont été étudiées dans la littérature, telle que l'élimination de variables approchée et la propagation itérative dans un graphe de jointure [17], ou bien dans le contexte voisin de l'inférence Bayésienne, la propagation de croyance itérative et la méthode de suppression d'arêtes [4][2]. Ces approches ont le défaut de ne

[1] En fait, cela dépend de la largeur d'arbre des sous-problèmes, qui est bornée par la taille de la plus grande clique dans le problème complet. Dans le cas de graphes peu denses, nous nous attendons à ce que cette taille soit petite, ce qui est montré dans nos expérimentations. En pratique, notre méthode d'approximation ne sera efficace que si le CSP à résoudre ne contient pas de contraintes globales.

[2] Cette méthode commence par résoudre un sous-problème structuré en poly-arbre, par la suite augmenté par la restauration d'arêtes supprimées, jusqu'à au final résoudre le problème complet. A l'in-

pas exploiter la structure locale (aussi appelée *micro-structure*) des instances comme c'est le cas pour BTD, grâce à son emploi d'une cohérence locale et d'un ordre de choix de variables dynamique au sein des clusters.

Dans la section suivante, nous introduisons les notations et rappelons les principes de l'algorithme BTD. La section 3 décrit notre version modifiée de BTD. Dans la section 4, nous introduisons la méthode d'approximation ApproxBTD. Des résultats expérimentaux sur des CSPs aléatoires et des benchmarks SAT sont présentés dans la section 5. Enfin, nous donnons une conclusion dans la section 6.

2 Préliminaires

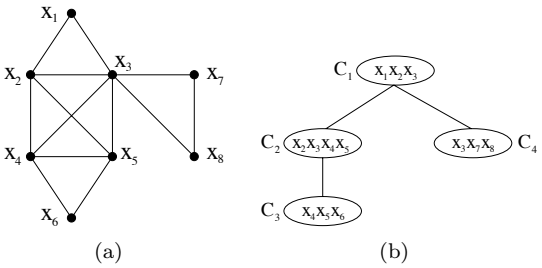
Un *problème de satisfaction de contraintes* (CSP) est défini par un tuple (X, D, C, R). X est un ensemble $\{x_1, \ldots, x_n\}$ de n variables. Chaque variable x_i prend ses valeurs dans le domaine fini d_{x_i} de D. Les variables sont soumises à un ensemble C de m contraintes. Chaque contrainte c est définie comme un ensemble $\{x_{c_1}, \ldots, x_{c_k}\}$ de variables. Une relation r_c (de R) est associée à chaque contrainte c telle que r_c représente l'ensemble des tuples autorisés sur $d_{x_{c_1}} \times \cdots \times d_{x_{c_k}}$. Notons que nous pouvons également définir les contraintes par des fonctions usuelles ou des prédicats, par exemple. Etant donné $Y \subseteq X$ tel que $Y = \{x_1, \ldots, x_k\}$, une *affectation* des variables de Y est un tuple $\mathcal{A} = (v_1, \ldots, v_k)$ sur $d_{x_1} \times \cdots \times d_{x_k}$. Une contrainte c est *satisfaite* par $\mathcal{A}$ si $c \subseteq Y, (v_1, \ldots, v_k)[c] \in r_c$, elle est dite *violée* sinon. Nous écrirons l'affectation $(v_1, \ldots, v_k)$ sous la forme plus explicite $(x_1 \leftarrow v_1, \ldots, x_k \leftarrow v_k)$. La structure d'un CSP peut être représentée par le graphe (X, C), appelé le *graphe de contraintes*, dont les sommets sont les variables de l'ensemble X et il y a une arête entre deux sommets s'il existe une constrainte entre les variables correspondantes.

Dans [16] une nouvelle méthode est proposée pour résoudre les CSP. Cette méthode appelée BTD (pour Backtracking with Tree-Decomposition) est une méthode énumérative guidée par une décomposition arborescente du graphe de contraintes. Une *décomposition arborescente* [22] d'un graphe $G = (X, E)$ est une paire $(\mathcal{C}, \mathcal{T})$ avec $\mathcal{T} = (I, F)$ un arbre et $\mathcal{C} = \{\mathcal{C}_i : i \in I\}$ une famille de sous-ensemble de X, tels que chaque cluster $\mathcal{C}_i$ est un nœud de $\mathcal{T}$ et vérifie : (1) $\cup_{i \in I} \mathcal{C}_i = X$, (2) Pour chaque arête $\{x, y\} \in E$, il existe $i \in I$ avec $\{x, y\} \subseteq \mathcal{C}_i$, (3) Pour tout $i, j, k \in I$, si k est sur le chemin de i à j dans $\mathcal{T}$, alors $\mathcal{C}_i \cap \mathcal{C}_j \subseteq \mathcal{C}_k$. La largeur d'arbre d'une décomposition arborescente $(\mathcal{C}, \mathcal{T})$ est égale à $\max_{i \in I} |\mathcal{C}_i| - 1$. La largeur d'arbre (tree-width) de G est la largeur minimale sur toutes les décompositions arborescentes de G. Notons que trouver une décomposition arborescente optimale est un problème NP-dur [1]. Cependant, nous pouvons facilement calculer un bonne décomposition arborescente en utilisant la notion de *graphes triangulés*. Une décomposition arborescente est calculée par triangulation (*i.e* ajout d'arêtes au) du graphe de contraintes pour qu'il devienne *triangulé*[3] afin d'y rechercher les cliques maximales dans le graphe de contrainte triangulé. La figure 1(b) présente une décomposition arborescente possible pour le graphe de la figure 1(a). Nous avons $\mathcal{C}_1 = \{x_1, x_2, x_3\}$, $\mathcal{C}_2 = \{x_2, x_3, x_4, x_5\}$, $\mathcal{C}_3 = \{x_4, x_5, x_6\}$ et $\mathcal{C}_4 = \{x_3, x_7, x_8\}$, et la largeur d'arbre est de 3. Dans la suite, à partir d'une décompostion arborecente, nous considérons un arbre enraciné (I, F) où $\mathcal{C}_1$ est la racine, nous noterons $Fils(\mathcal{C}_i)$ l'ensemble des clusters fils de $\mathcal{C}_i$ et $Desc(\mathcal{C}_j)$ l'ensemble des variables qui appartiennent à $\mathcal{C}_j$ ou à un descendant $\mathcal{C}_k$ de $\mathcal{C}_j$ dans l'arbre enraciné en $\mathcal{C}_j$. Par exemple, $Desc(\mathcal{C}_2) = \mathcal{C}_2 \cup \mathcal{C}_3 = \{x_2, x_3, x_4, x_5, x_6\}$.

La première étape de BTD consiste à calculer une décomposition arborescente du graphe de contraintes. Cette décomposition arborescente permet d'obtenir un ordre partiel sur les variables permettant à BTD d'exploiter les propriétés structurelles du graphe et de couper dans l'arbre de recherche. En effet, les variables

verse, ApproxBTD commence directement avec un sous-problème triangulé pouvant être plus grand que le poly-arbre de la méthode précédente.

[3] Un graphe est triangulé si chaque cycle de longueur supérieur à quatre a une corde, *i.e* une arête reliant deux sommets non consécutifs dans le cycle.



(a) (b)

FIG. 1 – (a) Un graphe de contraintes sur 8 variables. (b) Une décomposition arborescente de ce graphe de contraintes.

sont affectées selon une recherche en profondeur d'abord à partir de la racine de la décomposition. Autrement dit, nous affectons les variables du cluster racine $\mathcal{C}_1$, puis celles de $\mathcal{C}_2$, celles de $\mathcal{C}_3$ etc... Par exemple, $x_1, x_2, \ldots, x_8$ est un ordre possible. De plus, la décomposition arborescente et l'ordre sur les variables permettent à BTD de diviser le problème $\mathcal{P}$ en plusieurs sous-problèmes. Etant donnés deux clusters $\mathcal{C}_i$ et $\mathcal{C}_j$ (avec $\mathcal{C}_j$ un fils de $\mathcal{C}_i$), le sous-problème enraciné en $\mathcal{C}_j$ dépend de l'affectation courante $\mathcal{A}$ sur $\mathcal{C}_i \cap \mathcal{C}_j$. On notera ce sous-problème $\mathcal{P}_{\mathcal{A},\mathcal{C}_i/\mathcal{C}_j}$. Son ensemble des variables est $Desc(\mathcal{C}_j)$. Le domaine de chaque variable appartenant à $\mathcal{C}_i \cap \mathcal{C}_j$ est réduit à sa valeur associée dans $\mathcal{A}$. Concernant l'ensemble des contraintes, il contient les contraintes qui impliquent au moins une variable apparaissant exclusivement dans $\mathcal{C}_j$ ou un de ses descendants. Considérons, par exemple, le CSP dont le graphe de contraintes est donné par la figure 1(a). Chaque domaine est $\{a, b, c, d\}$ et chaque contrainte $c_{ij} = \{x_i, x_j\}$ a une relation $r_{c_{ij}}$ telle que $x_i \neq x_j$. Soit $\mathcal{A} = (x_2 \leftarrow b, x_3 \leftarrow c)$, l'ensemble de variables de $\mathcal{P}_{\mathcal{A},\mathcal{C}_1/\mathcal{C}_2}$ est $Desc(\mathcal{C}_2)$, (avec $d_{x_2} = \{b\}, d_{x_3} = \{c\}$ et $d_{x_4} = d_{x_5} = d_{x_6} = \{a, b, c, d\}$) et son ensemble de contraintes est $\{c_{24}, c_{25}, c_{34}, c_{35}, c_{45}, c_{46}, c_{56}\}$. Nous définissons la notion de *goods structurels*. Un good structurel de $\mathcal{C}_i$ par rapport à $\mathcal{C}_j$ (avec $\mathcal{C}_j$ un fils de $\mathcal{C}_i$) est l'affectation courante $\mathcal{A}$ sur $\mathcal{C}_i \cap \mathcal{C}_j$ pouvant être étendue sur le sous-problème $\mathcal{P}_{\mathcal{A},\mathcal{C}_i/\mathcal{C}_j}$. Par exemple, si nous considérons l'affectation $\mathcal{A} = (x_2 \leftarrow b, x_3 \leftarrow c)$ sur $\mathcal{C}_1 \cap \mathcal{C}_2$ nous obtenons le good $(\mathcal{A})$ grâce à l'affectation sur $Desc(\mathcal{C}_2)$ défini par $(x_2 \leftarrow b, x_3 \leftarrow c, x_4 \leftarrow a, x_5 \leftarrow d, x_6 \leftarrow b)$ satisfaisant toutes les contraintes de $\mathcal{P}_{\mathcal{A},\mathcal{C}_1/\mathcal{C}_2}$. Dans un premier temps cette affectation $(\mathcal{A})$ est explorée, puis lorsque son extension sur $\{x_4, x_5, x_6\}$ est valide, $(\mathcal{A})$ est enregistrée comme un good. Ainsi, si durant la recherche, l'affectation $\mathcal{A} = (x_2 \leftarrow b, x_3 \leftarrow c)$ est à nouveau étudiée, la recherche n'explore pas le sous-problème $\mathcal{P}_{\mathcal{A},\mathcal{C}_1/\mathcal{C}_2}$ car nous avons déjà prouvé qu'il existe une solution compatible avec $\mathcal{A}$. Au contraire, si aucune solution n'est trouvée pour une autre affectation $\mathcal{A}'$ sur $\mathcal{P}_{\mathcal{A}',\mathcal{C}_1/\mathcal{C}_2}$, un *nogood structurel* est alors enregistré. Ce nogood pourra être utilisé comme une nouvelle contrainte du problème.

Dans la section suivante, nous définirons la notion de good structurel pour le comptage, qui est basée sur les mêmes principes que les goods et nogoods structurels.

3 Dénombrement de solutions avec BTD

La méthode BTD est une adaptation de BTD pour le comptage de solutions. Comme pour BTD, la première étape de BTD consiste à calculer une décomposition arborescente du graphe de contraintes. Cette décompostion arborescente induit un ordre partiel sur les variables pour exploiter les propriétés structurelles du graphe et permettre d'élaguer l'arbre de recherche. Tandis que BTD utilise la notion de good, pour le dénombrement de solutions nous utilisons la notion de *#good*. Le but est de sauver le nombre de solutions des sous-problèmes induit par la décomposition arborescente. En effet, un # good de $\mathcal{C}_i$ par rapport à $\mathcal{C}_j$ (avec $\mathcal{C}_j$ un fils de $\mathcal{C}_i$) est une paire $(\mathcal{A}, nb)$ où $\mathcal{A}$ est une affectation $\mathcal{C}_i \cap \mathcal{C}_j$ et nb le nombre de solutions du sous-problème $\mathcal{P}_{\mathcal{A},\mathcal{C}_i/\mathcal{C}_j}$. Dans l'exemple de la Section 2, si nous considérons l'affectation $\mathcal{A} = (x_4 \leftarrow a, x_5 \leftarrow d)$ sur $\mathcal{C}_2 \cap \mathcal{C}_3$ nous obtenons un #good $(\mathcal{A}, 2)$ car nous avons deux solutions pour le sous-problème $\mathcal{P}_{\mathcal{A},\mathcal{C}_2/\mathcal{C}_3}$.

BTD explore l'espace de recherche suivant l'ordre des variables induit par la décomposition arborescente. Ainsi, on commence par les

Algorithme 1 : BTD($\mathcal{A}$, $\mathcal{C}_i$, $V_{\mathcal{C}_i}$) : entier

```
if V_{C_i} = ∅ then
    if Fils(C_i) = ∅ then
        return 1
    else
        F ← Fils(C_i)
        NbSol ← 1
        while F ≠ ∅ et NbSol ≠ 0 do
            Choisir C_j in F
            F ← F − {C_j}
            if (A[C_i ∩ C_j], nb) n'est pas un #good
            dans P then
                nb ← BTD(A, C_j, V_{C_j} − (C_i ∩ C_j))
                enregistre le #good
                (A[C_i ∩ C_j], nb) de C_i/C_j dans P
            NbSol ← NbSol × nb ;
        return NbSol
else
    Choisir x ∈ V_{C_i}
    NbSol ← 0
    d ← d_x
    while d ≠ ∅ do
        Choisir v dans d
        d ← d-{x}
1       if A ∪ {x ← v} ne viole aucune c ∈ C then
            NbSol ← NbSol+BTD(A ∪ {x ← v},
            C_i, V_{C_i}-{x})
    return NbSol
```

variables du cluster racine $\mathcal{C}_1$. A l'intérieur du cluster $\mathcal{C}_i$, on affecte une valeur à une variable, on "backtrack" si une contrainte est violée. Ce schéma peut être amélioré avec le maintient de l'arc consistance. Lorsque toutes les variables de $\mathcal{C}_i$ sont affectées, BTD calcule le nombre de solutions du sous-problème induit par le premier fils de $\mathcal{C}_i$, s'il en existe un. Plus généralement, considérons $\mathcal{C}_j$ un fils de $\mathcal{C}_i$. Etant donnée une affectation courante $\mathcal{A}$ sur $\mathcal{C}_i$, BTD vérifie si l'affectation $\mathcal{A}[\mathcal{C}_i \cap C_j]$ correspond à un #good. Si c'est le cas, BTD multiplie le nombre de solutions enregistré avec le nombre de solutions de $\mathcal{C}_i$ avec $\mathcal{A}$ comme affectation. Sinon, on étend $\mathcal{A}$ sur $Desc(\mathcal{C}_i)$ dans l'ordre pour compter le nombre nb d'extensions consistantes et on enregistre le #good $(\mathcal{A}[\mathcal{C}_i \cap \mathcal{C}_j], nb)$. BTD calcule le nombre de solutions du sous-problème induit par le fils suivant de $\mathcal{C}_i$. Finalement lorsque chaque fils de $\mathcal{C}_i$ a été examiné, BTD essaye de modifier l'affectation courante de $\mathcal{C}_i$. Le nombre de solutions de $\mathcal{C}_i$ est la somme des solutions de chaque affectation.

BTD est décrit par l'algorithme 1. Etant donnée une affectation $\mathcal{A}$ et un cluster $\mathcal{C}_i$, BTD regarde le nombre d'extensions $\mathcal{B}$ de $\mathcal{A}$ sur $Desc(\mathcal{C}_i)$ tel que $\mathcal{A}[\mathcal{C}_i\text{-}V_{\mathcal{C}_i}] = \mathcal{B}[\mathcal{C}_i\text{-}V_{\mathcal{C}_i}]$. $V_{\mathcal{C}_i}$ est l'ensemble des variables non affectées de $\mathcal{C}_i$. Le premier appel est BTD($\emptyset, \mathcal{C}_1, \mathcal{C}_1$) et il retourne le nombre de solutions. Notons $\mathcal{S}_{\mathcal{A},\mathcal{C}_i/\mathcal{C}_j}$ le nombre de solutions d'un sous-problème $\mathcal{P}_{\mathcal{A},\mathcal{C}_i/\mathcal{C}_j}$. Ainsi, $\mathcal{S}_{\mathcal{A},\mathcal{C}_i/\mathcal{C}_j} = \sum_{\mathcal{B}} \mathcal{S}_{\mathcal{B},\mathcal{C}_i/\mathcal{C}_j}$ pour chaque affectation $\mathcal{B}$ sur $Desc(\mathcal{C}_j)$ tel que $\mathcal{A}[\mathcal{C}_i \cap \mathcal{C}_j] = \mathcal{B}[\mathcal{C}_i \cap \mathcal{C}_j]$. Par exemple, avec $\mathcal{A} = (x_1 \leftarrow a, x_2 \leftarrow b, x_3 \leftarrow c)$ comme affectation de C_1 alors $\mathcal{S}_{\mathcal{A},\mathcal{C}_1/\mathcal{C}_2} = \mathcal{S}_{\mathcal{A}\cup(x_4\leftarrow a,x_5\leftarrow d),\mathcal{C}_1/\mathcal{C}_2} + \mathcal{S}_{\mathcal{A}\cup(x_4\leftarrow d,x_5\leftarrow a),\mathcal{C}_1/\mathcal{C}_2} = 2 + 2 = 4$. Soit $\mathcal{V}ar(\mathcal{P}_{\mathcal{A},\mathcal{C}_i/\mathcal{C}_j})$ l'ensemble des variables de $\mathcal{P}_{\mathcal{A},\mathcal{C}_i/\mathcal{C}_j}$ moins les variables de $\mathcal{C}_i \cap \mathcal{C}_j$. Autrement dit, c'est l'ensemble des variables de $\mathcal{P}_{\mathcal{A},\mathcal{C}_i/\mathcal{C}_j}$ non affectées. Si $\mathcal{C}_i$ a k fils : $\mathcal{C}_{i_1}, ..., \mathcal{C}_{i_k}$ alors $\cap_{1\leqslant j\leqslant k} \mathcal{V}ar(\mathcal{P}_{\mathcal{A},\mathcal{C}_i/\mathcal{C}_{i_j}}) = \emptyset$. On note $\mathcal{S}_{\mathcal{A},\mathcal{C}_i}$ le nombre de solutions de $Desc(\mathcal{C}_i)$ avec l'affectation $\mathcal{A}$ sur $\mathcal{C}_i$. D'où, $\mathcal{S}_{\mathcal{A},\mathcal{C}_i} = \prod_{1\leqslant j\leqslant k} \mathcal{S}_{\mathcal{A},\mathcal{C}_i/\mathcal{C}_{i_j}}$. Par exemple, le nombre de solutions de $\mathcal{P}_{\mathcal{A},\mathcal{C}_1/\mathcal{C}_2}$ (avec l'affectation $\mathcal{A} = (x_1 \leftarrow a, x_2 \leftarrow b, x_3 \leftarrow c)$ est 4 sur (x_4, x_5, x_6) et 6 solutions pour $\mathcal{P}_{\mathcal{A},\mathcal{C}_1/\mathcal{C}_4}$ sur (x_7, x_8). Donc, il y a 24 solutions sur $(x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8)$ avec $\mathcal{A}$. Notons que pour $[\mathcal{C}_1 \cap \mathcal{C}_4]$, $((x_3 \leftarrow c), 6)$ est un #good. Pour une autre affectation sur $\mathcal{C}_1$, e.g $\mathcal{A}' = (x_1 \leftarrow b, x_2 \leftarrow a, x_3 \leftarrow c)$, il n'est pas nécessaire de calculer les solutions sur $\mathcal{C}_4$ car le #good $((x_3 \leftarrow c), 6)$ peut être utilisé pour $\mathcal{A}'$.

La complexité en espace de BTD est $O(n.s.d^s)$ et celle en temps est $O(n.m.d^{w+1})$ avec $w+1$ la taille du plus grand cluster $\mathcal{C}_k$ et s la taille de la plus grande intersection $\mathcal{C}_i \cap \mathcal{C}_j$ ($\mathcal{C}_j$ un fils de $\mathcal{C}_i$).

En pratique, pour les problèmes ayant une grande largeur d'arbre, BTD explose en temps et en mémoire, voir à la Section 5. Dans ce cas, nous sommes intéressés par une méthode d'approximation.

4 Approximation avec BTD

Nous considérons ici des CSPs qui ne sont pas nécessairement structurés. Pour les résoudre, nous proposons d'exploiter BTD en définissant une nouvelle méthode d'approximation appelée ApproxBTD.

Sans perte de généralité, nous considérons le

cas de CSP binaires dans la présentation de la méthode. Nous pouvons définir une collection de sous-problèmes en partitionnant l'ensemble des contraintes, c'est à dire dans le cas de CSP binaires, en partitionnant l'ensemble des arêtes du graphe de contraintes. Nous remarquons que chaque graphe (X, C) peut être partitionné en k sous-graphes (X_1, E_1), ... , (X_k, E_k) tels que $\cup X_i = X$, $\cup E_i = C$ et $\cap E_i = \emptyset$ et tel que chaque sous-graphe (X_i, E_i) soit triangulé. Ainsi, chaque (X_i, E_i) peut être associé à un sous-problème structuré $\mathcal{P}_i$ (avec l'ensemble de variables X_i et l'ensemble de contraintes correspondant à E_i), qui peut être résolu efficacement en utilisant BTD. Ce partitionnement s'appuie sur le fait qu'il est facile de trouver une décomposition arborescente de largeur d'arbre minimum pour un graphe triangulé (*théorème de Fulkerson et Gross, 1965*). Supposons que $\mathcal{S}_{\mathcal{P}_i}$ est le nombre de solutions pour chaque sous-problème $\mathcal{P}_i$, $1 \leqslant i \leqslant k$. Nous estimerons le nombre de solutions de $\mathcal{P}$ en exploitant la propriété suivante. Premièrement, nous dénotons $\mathbb{P}_{\mathcal{P}}(\mathcal{A})$la probabilité de $\mathcal{A}$ *est une solution de* $\mathcal{P}$. $\mathbb{P}_{\mathcal{P}}(\mathcal{A}) = \dfrac{\mathcal{S}_{\mathcal{P}}}{\prod_{x \in X} d_x}$.

Propriété 1. *Soit un CSP donné* $\mathcal{P} = (X, D, C, R)$ *et une partition* $\{\mathcal{P}_1, ..., \mathcal{P}_k\}$ *de* $\mathcal{P}$ *induite par une partition de* C *en* k *éléments.*

$$\mathcal{S}_{\mathcal{P}} \approx \left\lceil \left(\prod_{i=1}^{k} \frac{\mathcal{S}_{\mathcal{P}_i}}{\prod_{x \in X_i} d_x} \right) \times \prod_{x \in X} d_x \right\rceil$$

Notons que l'approximation retourne une réponse exacte si tous les sous-problèmes sont indépendants ($\cap X_i = \emptyset$) ou $k = 1$ ($\mathcal{P}$ est déjà triangulé) ou si il existe un sous-problème incohérent $\mathcal{P}_i$. De plus, nous pouvons fournir un majorant trivial du nombre de solutions dû au fait que chaque sous-problème $\mathcal{P}_i$ est une relaxation de $\mathcal{P}$ (le même argument est utilisé dans [21] pour construire un majorant).

$$\mathcal{S}_{\mathcal{P}} \leq \min_{i \in [1,k]} \left\lceil \frac{\mathcal{S}_{\mathcal{P}_i}}{\prod_{x \in X_i} d_x} \times \prod_{x \in X} d_x \right\rceil$$

Notre méthode appelée ApproxBTD est décrite par l'algorithme 2. Appliqué sur un problème $\mathcal{P}$ avec un graphe de contraintes (X, C),

Algorithme 2 : ApproxBTD($\mathcal{P}$) : integer

```
Soit G' = (X', C') un graphe de contraintes associé à P ;
i ← 0 ;
while G' ≠ ∅ do
    i ← i + 1 ;
    Calculer un sous-graphe partiel triangulé (X_i, E_i) de G' ;
    Soit P_i le sous-problème associé à (X_i, E_i) ;
    S_{P_i} ←BTD(∅, C'_1, C'_1) avec C'_1 le cluster racine de la décomposition arborescente de P_i ;
    G' ← (X', C' − E_i) avec X' l'ensemble des variables induites par C'-E_i ;
k ← i ;
return ⌈ ∏_{i=1}^{k} S_{P_i} / ∏_{x∈X_i} d_x × ∏_{x∈X} d_x ⌉ ;
```

la méthode construit une partition $\{E_1, ..., E_k\}$ de C telle que le graphe de contraintes (X_i, E_i) est triangulé pour tout $1 \leqslant i \leqslant k$. Les sous-problèmes associés aux (X_i, E_i) sont résolus par BTD. La méthode retourne une approximation du nombre de solutions de $\mathcal{P}$ en utilisant la propriété 1.

Le nombre d'itérations de ApproxBTD est borné par n (nous avons au moins $n-1$ arêtes (un arbre) à chaque itération ou bien des sommets ont été effacés). Chaque sous-graphe triangulé et sa décomposition arborescente optimale associée peuvent être calculé en $O(nm)$ opérations [7][4]. De plus, nous garantissons que la largeur d'arbre w (plus un) de chaque sous-graphe triangulé produit est au plus égale à K, la taille de la plus grande clique (*clique number*) de $\mathcal{P}$. Soit w^* la largeur d'arbre (minimale) de $\mathcal{P}$, nous avons $w + 1 \leq K \leq w^* + 1$. Finalement, la complexité temporelle de ApproxBTD est $O(n^2 m d^K)$ et sa complexité mémoire est $O(nKd^{K-1})$.

5 Résultats expérimentaux

Nous avons effectué nos expérimentations sur des CSP aléatoires et sur des benchmarks SAT. Les expérimentations ont été réalisées sur

[4]Notons que cette approche retourne un sous-graphe maximal au sens de l'inclusion en nombre de contraintes pour un CSP binaire seulement. Dans le cas d'un CSP non-binaire (également en SAT), nous ne garantissons pas la maximalité du sous-graphe et ajoutant dans le sous-problème toutes les contraintes / clauses dont la clique associée d'arêtes est totalement incluse dans le sous-graphe.

un Pentium IV 3.2 GHz (resp. Xeon 2, 66 GHz) avec 1 Go (resp. 32 Go) pour les instances CSP (resp. SAT). Nous avons limité à une heure le temps autorisé pour résoudre chaque instance. Dans BTD à la ligne 1, nous utilisons Forward Checking au lieu de Backward Checking pour des raisons d'efficacité. A l'intérieur des clusters l'ordre dynamique *min domaine / max degré* est utilisé pour le choix de variables.

5.1 Instances aléatoires de CSP structurés

Nous étudions et comparons ApproxBTD avec BTD sur des instances structurées générées aléatoirement. Pour cela, nous considérons des instances aléatoires de k-arbres partiels avec les paramètres $(n, d, r_{max}, t, s_{max}, nc, nr)$ où chaque graphe de contraintes est un arbre de nc cliques et nr est le pourcentage d'arêtes supprimées dans les cliques. Il y a n variables avec un domaine de taille d, la taille de la plus grande clique est r_{max}, et la taille de la plus grande intersection est s_{max}. Chaque contrainte a une dureté égale à t, le pourcentage de tuples interdits parmi les tuples possibles pour la contrainte.

Les résultats présentés dans les Figures 2 et 3 sont les résultats du nombre de solutions et du temps CPU en millisecondes pour les classes $(50, 15, 8, t, 3, 10, 30\%)$ avec la dureté t variant entre 40% et 65%.

Chaque point dans la Figure 2 représente l'évaluation trouvée par ApproxBTD par rapport au nombre de solutions trouvé par BTD pour chaque instance. Nous observons que plus il y a de solutions, meilleure est l'approximation.

Le temps requis pour ces approximations est illustré par la Figure 3. Lorsque la dureté est inférieur à 53%, ApproxBTD est plus rapide que BTD. Pour les plus grandes duretés BTD peut être plus rapide que ApproxBTD car il résoud seulement un problème contre une collection de sous-problèmes.

5.2 Benchmarks SAT

Les benchmarks SAT viennent de `www.satlib.org`. Nous avons sélectionné les instances satisfaisables académiques (Tours de

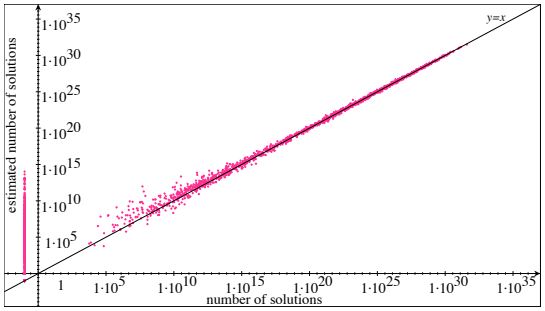


FIG. 2 – Comparaison entre le nombre de solutions trouvé par BTD et l'estimation trouvée par ApproxBTD sur les CSP structurés aléatoires (les points d'abscisses -0.1 représentent les instances inconsistantes).

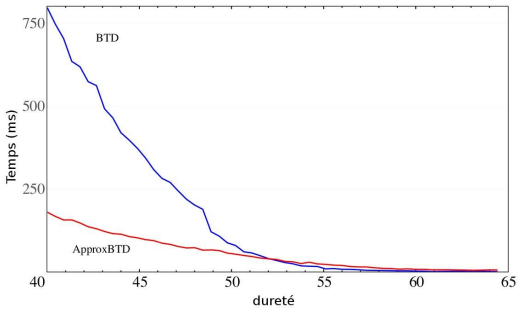


FIG. 3 – Temps CPU en millisecondes pour BTD et ApproxBTD sur les CSP structurés aléatoires.

Hanoi *hanoi*, All-Interval Series *ais*) et industrielles (analyse de faute de circuit *ssa* et *2bit*). Nous comparons BTD avec le compilateur d-DNNF `c2d` [6] qui exploite également la structure du problème. Ces deux méthodes utilisent l'ordre d'élimination de variables *MinFill* (sauf pour *hanoi* où nous avons utilisé l'ordre par défaut) pour construire une décomposition arborescente / d-DNNF. Nous avons également comparé ApproxBTD avec une méthode d'approximation nommée `SampleCount` [12] qui donne une estimation du minorant du nombre de solutions avec un intervalle de confiance élevé. La Table 1 résume nos résultats. Les colonnes sont le nom de l'instance, le nombre de

Instances	Vars	Clauses	w	Solutions	c2d	BTD	ApproxBTD			SampleCount	
					Temps	Temps	w	Solutions	Temps	Solutions	Temps
académiques											
ais6	61	581	41	24	0.04	0.06	5	≈ 1	0.09	≥ 6	0.184
ais8	113	1520	77	44	0.51	2.34	7	≈ 1	0.52	≥ 8	41.31
ais10	181	3151	116	296	17.14	390.32	9	≈ 1	2.69	≥ 38	384.8
ais12	265	4269	181	1328	1162.75	-	11	≈ 1	2.69	≥ 0	17.6
hanoi4	718	4934	46	1	3.41	1.72	6	≈ 1	1.57	≥ 0	5.26
hanoi5	1931	14468	58	1	-	25.46	7	≈ 1	14.98	≥ 0	6.19
industrielles											
ssa7552-038	1501	3575	25	2.84e40	0.15	0.72	7	$\approx$ 6.34e35	1.36	$\geq$ 6.67e38	1763
ssa7552-158	1363	3034	9	2.56e31	0.10	0.19	5	$\approx$ 2.59e27	0.79	$\geq$ 3.57e29	175
ssa7552-159	1363	3032	11	7.66e33	0.09	0.25	5	$\approx$ 1.09e30	0.84	$\geq$ 1.62e32	237
ssa7552-160	1391	3126	12	7.47e32	0.12	0.29	5	$\approx$ 2.88e33	0.99	$\geq$ 2.02e31	1117
2bitcomp_5	125	310	36	9.84e15	0.47	11.53	6	$\approx$ 2.23e15	0.02	$\geq$ 1.80e15	1.42
2bitmax_6	252	766	58	2.10e29	18.71	-	7	$\approx$ 1.98e28	0.1	$\geq$ 1.02e28	8.42

TAB. 1 – Solution counting for SAT instances. Time in seconds. A "-" means the instance was not solved in less than 1 hour.

variables booléennes, le nombre de clauses, la largeur d'arbre de la décomposition arborescente, le nombre exact de solutions, le temps CPU en secondes pour `c2d` et BTD, pour ApproxBTD : la largeur d'arbre maximale pour tous les sous-problèmes triangulés, le nombre de solutions approché et le temps, enfin pour `SampleCount` : le minorant du nombre de solutions et le temps. Nous remarquons que BTD peut résoudre des instances ayant une petite largeur d'arbre. `c2d` est généralement plus rapide (excepté pour *hanoi5*) mais il souffre également lorsqu'on a une grande largeur d'arbre (e.g. *ais*). Notre méthode d'approximation ApproxBTD exploite une partition du graphe de contraintes tels que les sous-problèmes résultants aient une petite largeur d'arbre ($w \leq 11$) comme le montrent les résultats. En pratique la méthode permet d'obtenir des résultats relativement rapidement même si la largeur d'arbre originale est importante[5]. La qualité de l'appoximation trouvée par ApproxBTD est relativement bonne et est comparable à `Samplecount` qui prend plus de temps et requiert un réglage de paramètres (nous utilisons les paramètres avec $t = 7, s = 20, \alpha = 1$ les options par défaut). Le majorant calculé par ApproxBTD n'est pas mentionné car il est très supérieur au nombre de solutions sur ces instances.

[5]Une petite largeur d'arbre ne permet pas en pratique d'être rapide à chaque fois (voir e.g. les instances ssa)

6 Discussion and conclusion

Dans cet article, nous avons proposé deux méthodes pour résoudre le problème de comptage du nombre de solutions d'un CSP. Ces méthodes exploitent une décomposition arborescente des CSPs. Nous avons présenté une méthode exacte qui est adaptée aux problèmes ayant une petite largeur d'arbre. Pour des problèmes avec une largeur d'arbre importante et un graphe de contraintes peu dense, nous avons présenté une nouvelle méthode d'approximation dont la qualité des résultats obtenus est proche des méthodes existantes mais qui sont obtenus bien plus rapidement et sans avoir à effectuer des réglages complexes de paramètres, à l'exception du choix de l'heuristique de décomposition arborescente.

Remerciements

Ce travail est partiellement supporté par le projet ANR STAL-DEC-OPT.

Références

[1] S. Arnborg, D. Corneil, and A. Proskurowski. Complexity of finding embeddings

in a k-tree. *SIAM Journal of Discrete Mathematics*, 8 :277–284, 1987.

[2] R. Bayardo and J. Pehoushek. Counting models using connected components. In *AAAI-00*, pages 157–162, 2000.

[3] R. Burton and J. Steif. Nonuniqueness of measures of maximal entropy for subshifts of finite type. In *Ergodic theory and dynamical system*, 1994.

[4] A. Choi and A. Darwiche. An edge deletion semantics for belief propagation and its practical impact on approximation quality. In *Proc. of AAAI*, pages 1107–1114, 2006.

[5] A. Darwiche. On the tractable counting of theory models and its applications to truth maintenance and belief revision. *Journal of Applied Non-classical Logic*, 11 :11–34, 2001.

[6] A. Darwiche. New advances in compiling cnf to decomposable negation normal form. In *Proc. of ECAI*, pages 328–332, 2004.

[7] P.M Dearing, D.R. Shier, and D.D. Warner. Maximal chordal subgraphs. *Discrete Applied Mathematics*, 20(3) :181–190, 1988.

[8] R. Dechter and R. Mateescu. The impact of and/or search spaces on constraint satisfaction and counting. In *Proc. of CP*, pages 731–736, Toronto, CA, 2004.

[9] R. Dechter and R. Mateescu. And/or search spaces for graphical models. *Artif. Intell.*, 171(2-3) :73–106, 2007.

[10] V. Gogate and R. Dechter. Approximate counting by sampling the backtrack-free search space. In *Proc. of AAAI-07*, pages 198–203, Vancouver, CA, 2007.

[11] V. Gogate and R. Dechter. Approximate solution sampling (and counting) on and/or search spaces. In *Proc. of CP-08*, pages 534–538, Sydney, AU, 2008.

[12] C. P. Gomes, J. Hoffmann ans A. Sabharwal, and B. Selman. From sampling to model counting. In *Proc. of IJCAI*, pages 2293–2299, 2007.

[13] C. P. Gomes, A. Sabharwal, and B. Selman. Model counting : A new strategy for obtaining good bounds. In *Proc. of AAAI*, 2006.

[14] C P. Gomes, W-J. van Hoeve, A. Sabharwal, and B. Selman. Counting CSP solutions using generalized XOR constraints. In *Proc. of AAAI-07*, pages 204–209, Vancouver, BC, 2007.

[15] R. M. Haralick and G. L. Elliot. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14 :263–313, 1980.

[16] P. Jégou and C. Terrioux. Hybrid backtracking bounded by tree-decomposition of constraint networks. *Artificial Intelligence*, 146 :43–75, 2003.

[17] K. Kask, R. Dechter, and V. Gogate. New look-ahead schemes for constraint satisfaction. In *Proc. of AI&M*, 2004.

[18] L. Kroc, A. Sabharwal, and B. Selman. Leveraging belief propagation, backtrack search, and statistics for model counting. In *Proc. of CPAIOR-08*, pages 127–141, Paris, France, 2008.

[19] T.K Satish Kumar. A model counting characterization of diagnoses. In *Proc. of the 13th International Workshop on Principles of Diagnosis*, 2002.

[20] M. Mann, G. Tack, and S. Will. Decomposition during search for propagation-based constraint. In *CoRR abs/0712.2389 :*, 2007.

[21] G. Pesant. Counting solutions of CSPs : A structural approach. In *Proc. of IJCAI*, pages 260–265, 2005.

[22] N. Robertson and P.D. Seymour. Graph minors II : Algorithmic aspects of treewidth. *Algorithms*, 7 :309–322, 1986.

[23] D. Roth. On the hardness of approximate reasonning. *Artificial Intelligence*, 82(1-2) :273–302, 1996.

[24] D. Sabin and E. C. Freuder. Contradicting conventional wisdom in constraint satisfaction. In *Proceedings of the 2nd International Workshop on Principles and Practice of Constraint Programming*, number

874 in LNCS, Rosario, Orcas Island (WA), May 1994. Springer-Verlag.

[25] T. Sang, F. Bacchus, P. Beame, H. Kautz, and T. Pitassi. Combining component caching and clause learning for effective model counting. In *SAT-04*, Vancouver, Canada, 2004.

[26] L. G. Valiant. The complexity of computing the permanent. *Theoretical Computer Sciences*, 8 :189–201, 1979.

[27] W. Wei and B. Selman. A new approach to model counting. In *Proc. of SAT-05*, pages 324–339, St. Andrews, UK, 2006.

Stratégies hybrides pour des décompositions optimales et efficaces

Philippe Jégou [1] **Samba Ndojh Ndiaye** [2] **Cyril Terrioux** [1]

[1] LSIS - UMR CNRS 6168
Université Paul Cézanne (Aix-Marseille 3)
Avenue Escadrille Normandie-Niemen 13397 Marseille Cedex 20, France
[2] LIRIS - UMR CNRS 5205
Université Claude Bernard Lyon 1
43 boulevard du 11 novembre 1918 (Nautibus) 69622 Villeurbanne cedex, France
{philippe.jegou, cyril.terrioux}@univ-cezanne.fr samba-ndojh.ndiaye@liris.cnrs.fr

Résumé

Dans cet article, nous nous intéressons aux méthodes de résolution de réseaux de contraintes par des approches de décompositions structurelles. La *Hiérarchie des Contraintes Traitables* définie dans [9] propose un classement de ces méthodes par rapport à leur complexité temporelle théorique. Cependant, elle crée une très forte dépendance entre les algorithmes de calcul de décompositions et les techniques d'exploitation de ces dernières.

Nous faisons tomber ces cloisons par la définition de nouvelles méthodes basées sur des combinaisons d'approches Tree-Clustering [6], BTD [16], Tree-Decomposition [18] et Hypertree-Decomposition [9].

Nous démontrons leur intérêt d'un point de vue théorique grâce à des résultats portant sur leur complexité temporelle. En outre, cette analyse nous a permis d'enrichir la Hiérarchie des Contraintes Traitables. Finalement, nous justifions notre approche avec des résultats expérimentaux.

1 Introduction

Nous nous intéressons, dans ce papier, à l'efficacité théorique et pratique de méthodes de résolution de CSP exploitant les propriétés topologiques des réseaux de contraintes. Un CSP peut être vu comme un problème qui, étant donné un ensemble fini de variables X prenant leur valeur dans des domaines finis D, vérifie l'existence d'une affectation de ces variables satisfaisant simultanément un ensemble de contraintes C. Une telle affectation est une solution du CSP. De manière plus générale, le problème consiste à trouver une solution voire l'ensemble des solutions. Malheureusement, déterminer l'existence d'une solution est un problème NP-complet. Malgré tout, différentes classes d'algorithmes de résolution de CSP ont été proposées. D'une part, nous avons les algorithmes combinant la technique standard du backtracking et celle du filtrage des domaines tels que FC (Forward Checking [13]) ou MAC (Maintaining Arc Consistency [19]). Même si ces algorithmes sont relativement efficaces en pratique, leur complexité temporelle théorique est évaluée de manière classique par $O(S.m^n)$, avec S la taille du CSP considéré, n le nombre de variables et m la taille maximum des domaines des variables.

Différentes approches ont été proposées pour améliorer cette borne qui se révèle la pire possible. En particulier, les méthodes structurelles exploitent les propriétés topologiques qui existent fréquemment dans les problèmes réels. Généralement, elles reposent sur les propriétés d'une décomposition arborescente [18] ou d'une décomposition hyperarborescente [9] du réseau de contraintes qui capture sa structure et permet donc d'exprimer ses propriétés topologiques. Etant donnée une décomposition arborescente de largeur w, la complexité temporelle des meilleures approches structurelles, basées sur cette décomposition, est en $O(S.m^{w+1})$, avec la garantie d'avoir $w < n$ et dans beaucoup de cas $w \ll n$. De même, si on considère une décomposition hyperarborescente de largeur h, il est possible d'obtenir une complexité temporelle en $O(S.r^h)$, avec r la taille maximum des relations

(tables) associées aux contraintes. [9] a montré que la décomposition hyperarborescente est meilleure que la décomposition arborescente car $h \leq w$. En outre, les auteurs ont introduit un outil formel, la *Hiérarchie des Contraintes Traitables*, pour la comparaison de plusieurs méthodes structurelles. Cet outil théorique considère les classes d'instances de CSP pouvant être résolues en un temps polynomial. Il apparaît dès lors que la méthode structurelle basée sur la décomposition hyperarborescente (notée MHD) [9] est plus performante que le Tree-Clustering basé sur la décomposition arborescente (noté TC) [6] puisque la classe des instances traitables par MHD contient strictement celles traitables par TC.

D'un point de vue théorique, cette hiérarchie est un outil d'un très grand intérêt dans la mesure où elle permet de se reposer sur des critères fiables pour choisir la meilleure approche pour la résolution d'une instance donnée. Cependant, nous savons qu'il peut parfois exister une grande différence entre les performances théoriques et pratiques d'une même méthode. De ce fait, nous allons étendre notre étude de cette hiérarchie en y intégrant des aspects liés au comportement et à l'utilisation pratiques de ces méthodes. Nous pouvons déjà dire que les méthodes de résolution basées sur une décomposition ne sont généralement pas utilisées comme cela est supposé dans la hiérarchie. En effet, cette dernière ne considère que des décompositions optimales. Or, le calcul d'une décomposition optimale est souvent inaccessible car il nécessite un temps qui peut parfois dépasser largement le temps nécessaire à la résolution du problème sans décomposition. Ainsi, en pratique, on préfère généralement utiliser un algorithme heuristique (approximation des valeurs optimales de w ou h) pour trouver une décomposition. Une fois celle-ci obtenue, l'étape suivante consiste à résoudre le CSP décomposé. Dans la hiérarchie, il existe un lien rigide entre le type de la décomposition et la méthode qui exploite celle-ci. Dans cet article, nous faisons tomber ces barrières en créant une séparation entre la méthode de décomposition graphique et la méthode exploitant celle-ci pour résoudre le problème. Pour cela, nous définissons et étudions de nouvelles combinaisons d'approches dans lesquelles un type de décomposition (par exemple une décomposition hyperarborescente) peut être utilisé pour la résolution par une méthode initialement définie pour un autre type de décomposition (par exemple TC).

Ensuite, nous montrons que ce type de méthodes hybrides a un réel intérêt d'un point de vue théorique. En effet, nous obtenons des résultats de complexité originaux permettant d'enrichir la hiérarchie de [9]. Enfin, nous montrons de manière empirique, sur des benchmarks de la dernière compétition CSP (http ://www.cril.univ-artois.fr/CPAI08/), que notre approche semble bien adaptée pour comparer des méthodes structurelles par rapport à leur capacité à résoudre des CSP.

La section 2 rappelle des notations ainsi que des résultats sur la complexité de méthodes énumératives et structurelles. La section 3 introduit et étudie de nouvelles approches qui combinent différents types de décompositions et de méthodes d'exploitation, puis propose de nouvelles bornes de complexité qui nous permettent d'étendre la *Hiérarchie des Contraintes Traitables*. La section 4 présente une comparaison pratique des méthodes ainsi définies et la section 5 donne une conclusion.

2 Rappels

2.1 Notations

Un *problème de satisfaction de contraintes fini* ou *réseau de contraintes fini* (X, D, C, R) est défini par un ensemble fini de variables $X = \{x_1, \dots x_n\}$, un ensemble de domaines finis $D = \{d_1, \dots d_n\}$ (le domaine d_i contient toutes les valeurs possibles pour la variable x_i), et un ensemble C de contraintes sur les variables. Une contrainte $c_i \in C$ sur un sous-ensemble ordonné de variables, $c_i = (x_{i_1}, \dots x_{i_{a_i}})$ (a_i est appelé l'*arité* de la contrainte c_i), est définie par une relation associée $r_i \in R$ de combinaisons autorisées de valeurs des variables de c_i. Nous allons utiliser la même notation pour la contrainte c_i et l'ensemble des variables sur lesquelles elle porte. Nous notons a l'arité maximale des contraintes de C. Sans perte de généralité, nous supposons que chaque variable est contenue dans au moins une contrainte. Une solution de (X, D, C, R) est une affectation de toutes les variables qui satisfait toutes les contraintes. La structure d'un CSP peut être représentée par l'hypergraphe (X, C), appelé l'*hypergraphe de contraintes*.

Dans cet article (de même que dans [9]), nous supposons que les relations ne sont pas vides et peuvent être représentées par des tables comme dans la théorie des bases de données relationnelles. Ensuite, nous notons S la taille d'un CSP (qui vérifie $S \leq n.m + a.r.|C|$ avec $r = max\{|r_i| : r_i \in R\}$). Soient $Y = \{x_1, \dots x_k\}$ un sous-ensemble de X et $\mathcal{A}$ une affectation sur Y. $\mathcal{A}$ peut être vue comme un tuple $\mathcal{A} = (v_1, \dots v_k)$. La *projection* de $\mathcal{A}$ sur un sous-ensemble Y' de Y, notée $\mathcal{A}[Y']$, est la restriction de $\mathcal{A}$ aux variables de Y'. La projection de la relation r_i sur le sous-ensemble Y' de c_i est l'ensemble des tuples $r_i[Y'] = \{t[Y'] | t \in r_i\}$. La jointure des relations sera notée $\bowtie$. La jointure de $\mathcal{A}$ avec la relation r_i est $\mathcal{A} \bowtie r_i = \{t | t \text{ est un tuple sur } Y \cup c_i \text{ et } t[Y] = \mathcal{A} \text{ et } t[c_i] \in r_i\}$.

2.2 Complexité de l'énumération

L'approche basique pour la résolution de CSP repose sur la procédure classique appelée *Backtracking (BT)*. La complexité temporelle de cet algorithme de base est en $O(a.r.|C|.m^n)$ car le nombre de nœuds potentiels développés durant la recherche est m^n et en supposant qu'un test de contrainte $\mathcal{A}[c_i] \in r_i$ est réalisé en $O(a.r)$. Pour simplifier les notations, elle sera exprimée en $O(S.m^n)$. Généralement, cet algorithme n'est jamais utilisé car il est clairement inefficace en pratique. L'approche la plus classique pour améliorer BT est basée sur le filtrage. Le premier algorithme proposé pour ce filtrage est le Forward Checking (FC [13]). Il était initialement défini sur les CSPs binaires. De nombreuses extensions et généralisations de FC ont été proposées pour résoudre des CSPs non binaires ou pour exploiter des techniques de filtrage plus puissantes [13, 19, 2]. Une de ces extensions appelée nFC2 [2] comporte un niveau de filtrage qui semble être un bon compromis entre le coût de cette opération et le bénéfice qu'elle apporte dans la simplification de la résolution. La complexité de nFC2 est également bornée par $O(S.m^n)$ comme cela est indiqué dans [2]. Récemment, [15] a proposé une nouvelle borne qui considère r la taille maximum des relations (tables) associées aux contraintes. Il a été démontré que la complexité temporelle de nFC2 est en $O(S.r^k)$, où k est la taille de $|C'|$, un recouvrement minimum de X. Pour rappel, on dit que C' est un recouvrement minimum de X si C' est recouvrement de X ($C' \subset C$ et $\cup_{c_i \in C'} = X$) et il n'existe pas de recouvrement C'' tel que $|C''| < |C'|$. Ce résultat peut être étendu à tout autre algorithme qui maintient un niveau de filtrage au moins aussi puissant que celui de nFC2. Par exemple, il est valable pour nFCi ($i \geq 2$) et MAC ([19]).

2.3 Méthodes de résolution basées décomposition

La décomposition de réseau de contraintes a été introduite dans [6] avec le Tree-Clustering (TC). TC et d'autres méthodes basées sur cette approche (voir [3]) reposent sur la notion de décomposition arborescente de graphes. Néanmoins, étant donné un CSP n-aire, et de ce fait un hypergraphe de contraintes, nous pouvons continuer à exploiter cette notion grâce au *graphe primal* de ce dernier. Soit $H = (X, C)$ un hypergraphe, le graphe primal de H est le graphe $G = (X, A_C)$ avec $A_C = \{\{x, y\} \subset X : \exists c_i \in C \ t.q. \ \{x, y\} \subset c_i\}$. Ainsi, étant donné un CSP, nous considérons son graphe primal pour définir une décomposition arborescente associée à ce CSP.

Définition 1 *Une décomposition arborescente d'un graphe $G = (X, A_C)$ est une paire (E, T) où $T = (I, F)$ est un arbre avec un ensemble de nœuds I et d'arêtes F, et $E = \{E_i : i \in I\}$ une famille de sous-ensembles de X, telle que chaque sous-ensemble (appelé cluster) E_i est un nœud de T et vérifie :*
(i) $\cup_{i \in I} E_i = X$,
(ii) $\forall \{x, y\} \in A_C$, $\exists i \in I$ avec $\{x, y\} \subset E_i$, et
(iii) $\forall i, j, k \in I$, si k est sur le chemin entre i et j dans T, alors $E_i \cap E_j \subset E_k$.

La largeur w d'une décomposition arborescente (E, T) est égale à $max_{i \in I}|E_i| - 1$. La largeur arborescente w^* de G est la largeur minimale sur toutes ses décompositions arborescentes.

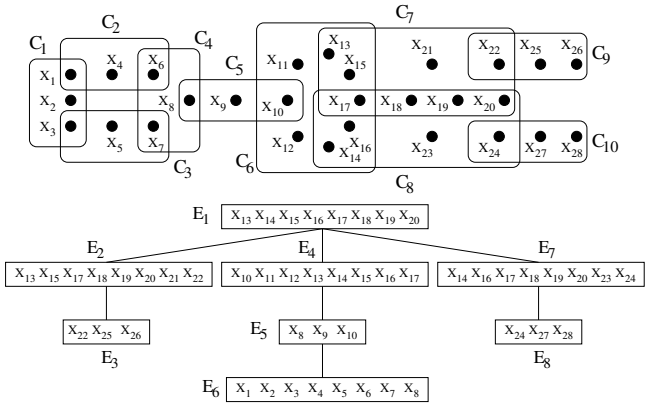


FIG. 1 – Un hypergraphe et une décomposition arborescente optimale.

La figure 1 présente un hypergraphe de contraintes et une décomposition arborescente possible de cet hypergraphe dont la largeur est minimum ($w^* = 7$).

Dans [6], le Tree-Clustering (noté ici TC-1989) a été défini initialement en utilisant un algorithme de complexité temporelle polynomiale (MCS [20]) pour calculer la décomposition arborescente. Plus précisément, la méthode se résume ainsi :

1. Calculer une décomposition arborescente du réseau de contraintes
   (a) Trianguler le graphe primal en utilisant MCS
   (b) Identifier les clusters de variables (les cliques maximales)
   (c) Construire un arbre de clusters (arbre de jointures)
2. Résoudre les sous-problèmes définis par les clusters de variables
3. Résoudre le problème acyclique résultant

La complexité de la première étape est limitée à $O(n^2)$. Cependant, nous n'avons aucune garantie sur l'optimalité du paramètre w. La complexité de la deuxième étape est en $O(S.m^{w+1})$. Ainsi, étant donné

un réseau de contraintes, plus la largeur de la décomposition est petite, plus cette complexité temporelle est petite. Or, la largeur de la décomposition arborescente calculée grâce à MCS peut être très éloignée de l'optimum. En outre, trouver une décomposition arborescente optimale est un problème NP-difficile. Nous faisons remarquer que dans [4] où TC-1989 est appelé Join Tree-Clustering, chaque sous-problème est défini par l'ensemble des variables du cluster et les contraintes dont les variables sont incluses dans ce dernier. Ce qui assure une complexité en $O(|C|.w.log(m).m^{w+1})$ pour la troisième étape. Finalement, le coût total de TC-1989 est en $O(S.m^{w+1})$.

Quant à la complexité en espace, celle-ci est liée à la sauvegarde des solutions des différents sous-problèmes qui vont conduire au nouveau problème acyclique. Elle est en $O(n.a.m^{w+1})$. Cependant, dans [4], Dechter suggère de mémoriser uniquement une partie des solutions, en fait leur projection sur les intersections entre les clusters. De ce fait, la complexité en espace est limitée à $O(n.a.m^s)$, où s est la taille maximum des intersections entre les clusters.

Défini initialement sur les CSPs binaires, TC-1989 a été étendu depuis lors aux CSPs n-aires de différentes manières [4]. Par ailleurs des améliorations de cette méthode ont été proposées pour remédier à certains de ses inconvénients (un espace mémoire requis trop important, résolution complète des sous-problèmes avant le test de la consistance globale du CSP, valeur de w qui peut être assez éloignée de l'optimum). Cependant, malgré ces améliorations, cette méthode demeure peu efficace en pratique. Par exemple, BTD [16] peut être considérée comme une approche efficace d'exploitation de décompositions arborescentes. Cette efficacité pratique vient du fait que BTD utilise un algorithme énumératif guidé par un ordre sur les variables induit par la décomposition arborescente. Cette approche évite généralement la résolution complète de tous les clusters. Comme pour TC-1989, la complexité temporelle de BTD est en $O(S.m^{w+1})$ alors que celle en espace est en $O(n.a.m^s)$. En outre, BTD considère une décomposition arborescente donnée qui peut être obtenue grâce à des algorithmes heuristiques ou exacts.

Alors que, TC-1989 peut être vue comme une méthode orientée variables puisque les clusters sont définis par des ensembles de variables, [9] propose une nouvelle méthode qui considère des clusters de contraintes. Cette approche est basée sur la notion de décomposition hyperarborescente qui est une généralisation de la décomposition arborescente.

Définition 2 *Etant donné un hypergraphe $H = (X, C)$, un hyperarbre de H est un triplet (T, χ, λ) où $T = (N, F)$ est arbre enraciné, et χ et λ sont des fonctions qui associent à chaque sommet $p \in N$ deux ensembles $\chi(p) \subset X$ et $\lambda(p) \subset C$. Si $T' = (N', F')$ est un sous-arbre de T, nous définissons $\chi(T') = \cup_{v \in N'} \chi(v)$. On note $sommets(T)$, l'ensemble N des sommets de T, et $racine(T)$ la racine de T. En outre, quelque soit $p \in N$, T_p est le sous-arbre de T enraciné en p.*

Définition 3 *Une décomposition hyperarborescente de H est un hyperarbre $HD = (T, \chi, \lambda)$ de H qui satisfait les conditions suivantes :*
(i) $\forall c \in C, \exists p \in sommets(T)$ t.q. $c \subset \chi(p)$,
(ii) $\forall x \in X$, l'ensemble $\{p \in sommets(T) : x \in \chi(p)\}$ induit un sous-arbre (connexe) de T,
(iii) $\forall p \in sommets(T), \chi(p) \subset \cup_{c \in \lambda(p)} c$,
(iv) $\forall p \in sommets(T), \cup_{c \in \lambda(p)} c \cap \chi(T_p) \subset \chi(p)$.

Une arête $c \in C$ est fortement couverte dans HD s'il existe $p \in sommets(T)$ tel que $c \subset \chi(p)$ et $c \in \lambda(p)$. Une décomposition hyperarborescente HD est une décomposition complète de H si chaque arête de H est fortement couverte dans HD.

La largeur h d'une décomposition hyperarborescente $HD = (T, \chi, \lambda)$ est $max_{p \in sommets(T)} |\lambda(p)|$. La largeur hyperarborescente h^* de H est la largeur minimum sur toutes ses décompositions hyperarborescentes.

Nous remarquons que les hypergraphes acycliques sont précisément ceux ayant une largeur hyperarborescente égale à 1. Pour la résolution de CSP, nous considérons uniquement des décompositions hyperarborescentes complètes. La figure 2 présente une décomposition hyperarborescente complète de l'hypergraphe de la figure 1.

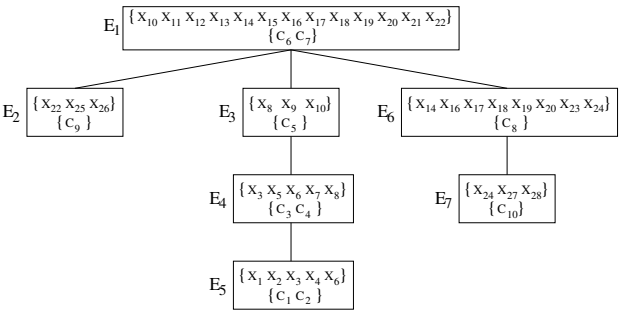


FIG. 2 – Une décomposition hyperarborescente optimale ($h^* = 2$).

La méthode basée sur cette notion de décomposition hyperarborescente (notée ici MHD-1999) a été également proposée dans [9]. Elle procède de la façon suivante :

1. Calculer une décomposition hyperarborescente du CSP
2. Résoudre chaque cluster par une jointure des relations associées aux contraintes du cluster
3. Résoudre le problème acyclique résultant

[9] présente une évaluation de la complexité, en supposant que la première étape peut être réalisée en $O(|C|^{2h^*}.n^2)$ grâce à l'algorithme *opt-k-decomp* défini dans [8]. Pour cela, l'hypergraphe considéré doit avoir une largeur hyperarborescente bornée par une constante. L'algorithme *opt-k-decomp* est donc capable de calculer une décomposition hyperarborescente optimale en un temps polynomial pour tout hypergraphe dont la largeur hyperarborescente est bornée par une constante. Le coût de la résolution des clusters dans la deuxième étape est borné par $O(S.r^h)$. La complexité en espace, en $O(r^h)$, est liée à la taille des relations résultant des jointures au sein des clusters. L'intérêt pratique de MHD-1999 n'a pas été clairement établi jusqu'à présent. Cela est probablement dû au manque d'algorithmes efficaces pour calculer une décomposition hyperarborescente, mais également à l'espace mémoire ($O(n.r^h)$) nécessaire dans la deuxième étape. Néanmoins, d'un point de vue théorique, cette méthode est clairement d'une très grande importance comme cela est démontré dans la *Hiérarchie des Contraintes Traitables* ([9]) (voir la figure 3). Cette hiérarchie donne une comparaison théorique des méthodes les plus connues pour la résolution de CSPs par décomposition d'(hyper)graphes de contraintes. Elle considère uniquement les méthodes D qui procèdent comme suit :

1. Reconnaître en un temps polynomial un CSP P traitable (en vérifiant si sa largeur notée D-*width* par rapport à la décomposition utilisée par D est bornée par une constante)
2. Calculer en un temps polynomial une décomposition de P dont la largeur est inférieure à cette constante
3. Transformer en un temps polynomial P en un CSP acyclique équivalent P'.
4. Résoudre P' en un temps polynomial (en $S^{D-width}$ puisque D-*width* est bornée par une constante).

Ainsi, si on considère MHD-1999, D-*width* $= h^*$. Tout CSP dont l'hypergraphe de contraintes a une largeur hyperarborescente bornée par une constante est traitable (peut être résolu en un temps polynomial) par MHD-1999.

Formellement, soient D_1 et D_2 deux méthodes de résolution basées décomposition. On dit que D_2 *généralise fortement* D_1 (représenté par un arc (D_1,D_2) dans la figure 3) si D_2 *généralise* D_1 (chaque problème traitable par D_1 est également traitable par D_2) et que D_2 *bat* D_1 (il existe une classe de problèmes traitables par D_2 et pas par D_1).

Alors que MHD-1999 est au sommet de la hiérarchie, il faut noter que formellement, TC-1989 ne devrait pas y figurer car cette méthode ne garantit pas l'utilisation d'une décomposition dont la largeur est bornée par la constante de la première étape. En effet, la largeur de la décomposition arborescente calculée par MCS peut être très éloignée de l'optimum et ainsi dépasser cette constante. Par ailleurs, il existe d'autres méthodes de résolution basées décomposition meilleures que MHD (en termes de complexité de résolution du CSP décomposé) : celle basée sur la décomposition hyperarborescente généralisée (MGHD [10]) et celle basée sur la décomposition hyperarborescente fractionnaire (MFHD [12]). Malheureusement ces approches ne peuvent figurer dans la hiérarchie car il n'existe pas d'algorithmes polynomiaux pour la reconnaissance des CSPs traitables dans ce cas, et pour le calcul des décompositions correspondantes. Ainsi, MHD demeure au sommet de la hiérarchie. Enfin, [7] a récemment proposé des outils algorithmiques intéressants pour calculer efficacement de bonnes approximations de décompositions hyperarborescentes optimales.

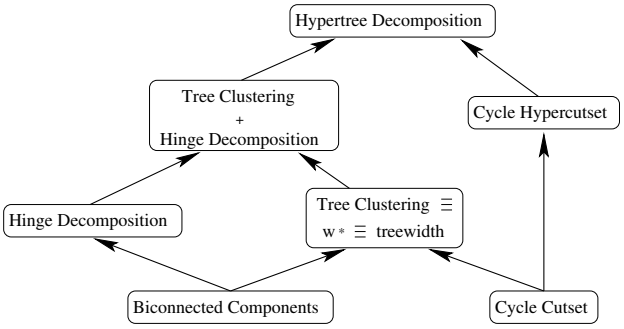


FIG. 3 – La hiérarchie des Contraintes Traitables.

3 De nouvelles méthodes pour résoudre des CSPs décomposés

3.1 Résolution VS Décompositions graphiques

En pratique, les méthodes de résolution basées décomposition ne sont pas véritablement utilisées comme cela est supposé dans la hiérarchie. Généralement, les deux premières étapes sont fusionnées, le calcul d'une décomposition dont la largeur est bornée par une constante permettant de conclure que la largeur du problème P est bornée par cette même constante. En plus, cette décomposition est calculée grâce à un algorithme heuristique à l'image de TC-1989 et non par un algorithme exact trop coûteux. En effet, de ce point de vue pratique, les valeurs optimales de w et h ne sont pas nécessaires voire même totalement inutiles [14]. Ensuite, les étapes 3 et 4 peuvent être considérées séparément comme dans TC-1989 et MHD-1999, ou fusionnées dans une unique étape à

l'image de BTD. Ainsi, dans cet article, nous considérons qu'une méthode de résolution basée décomposition notée DM_{DEC} a pour entrée un CSP P et une décomposition graphique (notée DEC) de P. Dès lors, la méthode DM résout P en utilisant la décomposition graphique considérée DEC. Par exemple, TC_{TD} est l'application de TC au CSP P en considérant une décomposition arborescente optimale tandis que TC_{MCS-TD} travaille avec une décomposition arborescente calculée par MCS pour résoudre P avec TC. De ce fait, TC_{MCS-TD} correspond à TC-1989 alors que TC_{TD} est le TC référencé dans la hiérarchie en considérant une décomposition arborescente optimale. Enfin, MHD_{HD} correspond à MHD-1999. Dans cet article, nous montrons l'intérêt de ce type d'approche par l'introduction de versions hybrides de méthodes $DM \in \{TC, MHD, BTD\}$ et de décompositions graphiques de type décomposition arborescente (TD) ou hyperarborescente (HD), optimales ou heuristiques.

Nous allons maintenant montrer comment des méthodes comme TC ou BTD, basées initialement sur des décompositions arborescentes, peuvent être étendues pour utiliser des décompositions hyperarborescentes, et inversement comment MHD peut être restreintes aux décompositions arborescentes.

3.2 De HD à TD

[15] donne une méthode pour transformer une décomposition hyperarborescente en une arborescente.

En effet, étant donnés un CSP $\mathcal{P} = (X, D, C, R)$ et $HD = (T, \chi, \lambda)$ une décomposition hyperarborescente de $H = (X, C)$ de largeur h, (T, χ) vérifie toutes les conditions requises pour être une décomposition arborescente à l'exception du fait qu'il peut exister un cluster $\chi(p)$ qui soit contenu dans un autre $\chi(p')$, avec $p, p' \in sommets(T)$. La définition suivante permet de construire cette décomposition arborescente grâce à la notion d'arbre de jointures d'un hypergraphe H [4] qui est un arbre connexe dont les nœuds sont les hyperarêtes de H telles que si un sommet x est dans deux nœuds alors x est dans tous les nœuds sur la chaîne reliant les deux nœuds.

Définition 4 *Etant donnée une décomposition hyperarborescente $HD = (T, \chi, \lambda)$ de $H = (X, C)$, $TD(HD) = (E, T')$ est définie par :*
(i) $E = \{\chi(p), p \in sommets(T) | \nexists p' \in sommets(T), \chi(p) \subsetneq \chi(p')\}$,
(ii) $T' = (I', F')$ est un arbre de jointures de l'hypergraphe (X, E).

Propriété 1 *[15] $TD(HD)$ est une décomposition arborescente de $H = (X, C)$.*

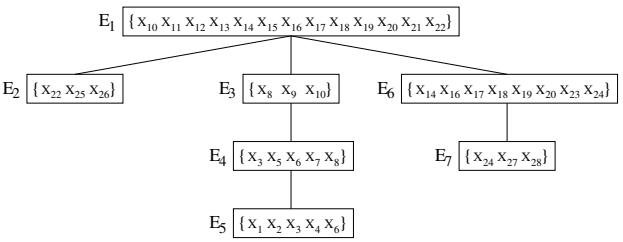


FIG. 4 – Décomposition arborescente induite (à partir de la figure 2).

Dans la figure 4, nous avons une décomposition arborescente qui n'est pas optimale ($w = 12$) et qui est induite par la décomposition hyperarborescente optimale de la figure 2.

3.3 De TD à (G)HD

Inversement, étant donnée une décomposition arborescente $TD = (E, T)$ de $H = (X, C)$, [15] propose une manière de calculer une décomposition hyperarborescente induite. Cette fois-ci, la construction n'est pas immédiate dans la mesure où recouvrir les différents clusters ne suffit pas car la quatrième condition de la définition de décomposition hyperarborescente n'est pas forcément vérifiée. De ce fait, il faut trouver des recouvrements des clusters satisfaisant cette condition supplémentaire. Ce qui peut être une tâche assez ardue. Or, cette condition est présente dans la définition dans le seul but de s'assurer de l'existence d'un algorithme permettant de calculer en un temps polynomial une décomposition hyperarborescente dont la largeur est bornée par une constante si elle existe. Dans notre cas, cet objectif est sans intérêt. Nous avons déjà une décomposition arborescente, et la meilleure décomposition hyperarborescente qu'elle induit est celle dont le recouvrement des clusters est minimum. Aussi, nous ne tiendrons plus compte de cette condition et allons construire une décomposition arborescente généralisée (GHD [10]).

Définition 5 *[10] Une décomposition hyperarborescente généralisée de H est un hyperarbre $HD = (T, \chi, \lambda)$ de H qui satisfait les conditions suivantes :*
(i) $\forall c \in C, \exists p \in sommets(T)$ t.q. $c \subset \chi(p)$,
(ii) $\forall x \in X$, l'ensemble $\{p \in sommets(T) : x \in \chi(p)\}$ induit un sous-arbre (connexe) de T,
(iii) $\forall p \in sommets(T), \chi(p) \subset \cup_{c \in \lambda(p)} c$,

La largeur gh d'une décomposition hyperarborescente généralisée $GHD = (T, \chi, \lambda)$ est $max_{p \in sommets(T)} |\lambda(p)|$. La largeur hyperarborescente généralisée gh^* de H est la largeur minimum sur toutes ses décompositions hyperarborescentes généralisées.

Nous associons un recouvrement minimum $\lambda(p)$ à chaque cluster $E_p \in E$ avec $p \in sommets(T)$.

Définition 6 *Etant donnée une décomposition arborescente $TD = (E, T)$ de $H = (X, C)$, nous définissons $GHD(TD) = (T, \chi, \lambda)$ comme suit :*
(i) $E = \chi$,
(ii) $\lambda = \{$recouvrement minimum de E_p dans $C, \forall p \in sommets(T)\}$.

Propriété 2 *$GHD(TD)$ est une décomposition hyperarborescente généralisée de largeur k, la taille maximum de $\lambda(p)$.*

Néanmoins, cette décomposition hyperarborescente généralisée n'est pas nécessairement complète. Pour la rendre complète, nous devons ajouter, pour toute arête c qui n'est pas fortement couverte, un fils à un nœud p tel que $c \subset E_p$. Soit $GHD_c(TD)$ la décomposition hyperarborescente généralisée complète obtenue de la sorte.

Théorème 1 *$GHD(TD)$ et $GHD_c(TD)$ ont la même largeur.*

La décomposition hyperarborescente généralisée induite par la décomposition arborescente de la figure 1 est facilement obtenue, et sa largeur est 4, parce que la taille maximum des $\lambda(p)$ vaut 4 avec le nœud associé à E_6 ($\{C_1, C_2, C_3, C_4\}$ est le recouvrement minimum).

En outre, si nous considérons gh^*, la largeur hyperarborescente généralisée d'un hypergraphe donné, nous savons que $gh^* \leq h^* \leq 3.gh^* + 1$ ([1]). Nous pouvons en déduire l'existence d'une décomposition hyperarborescente $HD_c(TD)$ de largeur au plus $3.k + 1$.

3.4 De Nouvelles Méthodes

Dorénavant, il est possible d'exploiter une décomposition hyperarborescente avec des méthodes de type TC ou BTD.

Auparavant, nous allons définir une nouvelle extension de TC (notée *TC-2009*) plus appropriée aux CSPs avec des contraintes n-aires. Etant donnés un CSP et une décomposition arborescente $TD = (E, T)$, le sous-problème associé au cluster E_i est défini, à l'image de TC-1989, par le même ensemble de variables E_i. Mais maintenant, l'ensemble des contraintes d'un cluster E_i est $C_{E_i} = \{c_j \in C : c_j \cap E_i \neq \emptyset\}$. Les relations associées à ces contraintes sont $R_{E_i} = \{r_j[c_j \cap E_i] : c_j \in C_{E_i}\}$. TC-2009 comporte également 3 étapes. La première calcule une décomposition arborescente du réseau de contraintes, en utilisant un algorithme de décomposition de (hyper)graphes alors que les deux étapes suivantes restent identiques. Ainsi, cette première étape est maintenant paramétrée par une décomposition graphique quelconque *DEC*. Si nous considérons une décomposition hyperarborescente optimale, nous définissons *TC-2009*$_{HD}$, qui exécute TC sur une décomposition arborescente $TD(HD)$ induite par HD et en considérant comme sous-problèmes, les clusters de variables et les contraintes dont l'ensemble des variables intersecte les clusters. De même, nous pouvons définir une large collection de méthodes à l'instar de *TC-2009*$_{TD}$ (TC basée sur une TD optimale), *TC-2009*$_{MCS(TD)}$ (TC basée sur une TD calculée par MCS), BTD_{HD} (BTD basée sur une HD optimale), BTD_{TD} (BTD basée sur une TD optimale), $BTD_{HMIN(HD)}$ (BTD basée sur une HD calculée grâce à une heuristique), $BTD_{HMIN(TD)}$ (BTD basée sur une TD calculée grâce à une heuristique), $BTD_{HMIN(TD)+HMAX(HD)}$ (BTD basée sur une TD optimale dont le nombre de contraintes dans les clusters a été maximisé grâce à une heuristique), etc.

Supposons que la largeur de décomposition hyperarborescente soit h dans le cadre de l'analyse de la complexité de *TC-2009*$_{HD}$. Chaque sous-problème (cluster) est résolu indépendamment (étape 2) en utilisant un algorithme de type nFC2. Grâce aux résultats présentés dans [15], le coût de la résolution d'un cluster E_i est maintenant en $O(S_i.r^{k_i})$, où S_i est la taille du sous-problème associé à E_i, et $k_i = k_{(E_i, C_{E_i})}$ (i.e. le paramètre associé au recouvrement minimum de E_i). Il faut noter que la taille de l'ensemble de solutions dans E_i est bornée par $O(r^{k_i})$. De ce fait, le coût total pour la résolution du CSP décomposé dans sa totalité est $O(S.r^k)$ où $k = max\{k_i : i \in I\}$. En plus, nous avons $k \leq h$. Nous pouvons donc établir que :

Théorème 2 *La complexité en temps de TC-2009*$_{HD}$ *et de BTD_{HD} est en $O(S.r^h)$.*

Ce résultat donne une présentation plus précise de la *Hiérarchie des Contraintes Traitables* puisque TC-2009$_{HD}$ et BTD$_{HD}$ sont au même niveau (sommet) dans la hiérarchie. Cela démontre que *TC-2009*$_{HD}$ est au moins aussi performante que MHD-1999. Plus précisément, la complexité temporelle de *TC-2009*$_{HD}$ et donc celle de BTD_{HD} sont identiques à celle de MHD-1999. Une autre conséquence de ce résultat est que nous disposons d'une nouvelle implémentation de MHD avec BTD_{HD} qui hérite de la même complexité en temps que MHD tout en limitant drastiquement la complexité en espace. Ainsi, cette approche peut potentiellement bénéficier d'une certaine efficacité pratique.

4 Expérimentations

Dans cette section, nous allons présenter les expérimentations qui ont été menées pour évaluer l'inté-

	FC	$BTD_{HMIN(TD)+HMAX(HD)}$		$BTD_{HMIN(TD)}$		$BTD_{HMIN(HD)}$			
CSP	temps	temps	w	temps	w	temps1	temps2	w	h
geo-1	18,01	14,19	26	18,03	19	MO	8,99	19	10
geo-9	10,40	19,00	26	5,46	18	TO	5,27	18	10
geo-12	14,18	17,50	31	17,68	22	MO	28,66	23	12
geo-19	17,04	30,64	23	170,28	18	TO	159,80	18	10
geo-41	11,46	11,12	24	8,03	19	12,93	8,09	19	10
geo-47	0	23,03	18	56,52	16	46,58	0,73	17	9
geo-52	10,96	10,91	31	7,51	19	8,70	7,44	19	10
geo-55	0,87	7,08	24	3,98	19	76,35	3,87	19	10
geo-57	5,61	6,69	26	6,19	23	4,68	6,16	23	12
geo-62	5,62	5,78	24	8,70	22	TO	11,93	20	11
geo-70	10,30	10,45	36	14,18	21	TO	13,96	21	11
geo-73	2,03	5,11	29	34,61	20	TO	8,17	21	11
geo-75	0,05	52,32	24	2,44	19	69,59	0,66	19	10
geo-86	12,81	13,50	27	15,97	23	TO	15,67	23	12
geo-94	16,82	14,40	27	18,68	18	TO	23,25	19	10
ren-3	TO	11,15	12	10,67	10	42,34	20,56	12	3
ren-6	TO	3,75	13	3,71	10	1279,55	2,70	13	3
ren-12	TO	11,85	12	11,64	10	134,24	10,49	10	3
ren-16	TO	10,36	12	6,04	10	3,65	6,43	13	3
ren-17	TO	5,42	12	9,59	10	145,06	3,41	11	3
ren-18	TO	12,09	11	12,23	11	39,34	10,46	12	3
ren-19	TO	12,07	11	7,74	9	49,25	16,36	11	3
ren-23	TO	2,81	11	12,87	9	28,82	3,79	12	4
ren-24	TO	8,05	14	7,97	11	35,01	7,63	12	4
ren-30	TO	9,81	13	3,80	10	202,05	8,25	11	3
ren-35	TO	12,28	12	7,32	10	57,33	13,19	11	3
ren-36	TO	1,78	13	3,80	11	225,76	4,88	13	4
ren-37	TO	17,01	15	13,68	12	13,64	21,16	14	4
ren-39	TO	35,55	16	13,45	12	746,24	1,79	12	5
ren-40	TO	8,60	14	5,86	11	65,55	9,01	12	4
ren-42	TO	3,44	15	2,48	12	TO	2,50	12	3
ren-47	TO	21,31	14	53,71	12	324,20	80,25	12	5

TAB. 1 – Temps de résolution (en s) des méthodes FC, $BTD_{HMIN(TD)+HMAX(HD)}$, $BTD_{HMIN(TD)}$ et $BTD_{HMIN(HD)}$, et paramètres des décompositions sur les instances des classes modifiedRenault (111 variables) et geom (50 variables) issues des benchmarks de la compétition CPAI08.

rêt pratique des méthodes que nous venons de définir. Les instances CSP que nous avons considérées sont issues de la compétition CPAI08 (http ://www.cril.univ-artois.fr/CPAI08/).

Les implémentations existantes de TC et MHD ne permettent pas de résoudre ces instances à cause de l'espace mémoire trop important qu'elles requièrent ou du temps beaucoup trop long qu'elles mettent pour résoudre séparément les sous-problèmes d'une décomposition. Ainsi, nous avons utilisé BTD qui a déjà montré son efficacité sur des CSPs structurés.

Par ailleurs, calculer une décomposition (hyper)arborescente optimale est un problème NP-difficile. La durée d'exécution des techniques exactes est trop importantes (voir [14]). En plus, il n'existe aucune garantie sur l'efficacité pratique de l'utilisation de ces décompositions. De ce fait, nous préférons des heuristiques avec une meilleure complexité temporelle pour calculer nos décompositions. Donc, nous ne considérerons ni BTD_{HD}, ni BTD_{TD}. Dans un premier temps, nous avons testé $BTD_{HMIN(HD)}$ et $BTD_{HMIN(TD)}$. Nous définissons $BTD_{HMIN(HD)}$ comme une extension de BTD qui gère les contraintes de manière analogue à MHD. Ainsi, lors de la résolution d'un cluster, seules les contraintes données par la HD sont prises en compte. En outre, pour calculer les HD, nous avons utilisé les heuristiques *Bucket Elimination for Hypertree* [7] et *det-k-decomp* [11] dont les implémentations sont disponibles sur internet (www.dbai.tuwien.ac.at/proj/hypertree/). Dans [11], elles sont évaluées comme les meilleures techniques pour calculer les HD. Dans $BTD_{HMIN(TD)}$, les TD sont calculées grâce à l'algorithme de triangulation Minfill qui est bien connu pour ses excellents résultats.

Les expérimentations ont été conduites sur un PC doté d'un Pentium IV de 3,2GHz et de 1Go de RAM, avec un système d'exploitation Linux. Pour chaque problème, la durée limite de résolution est bornée à 1800s. Au-delà, le problème est considéré comme non résolu et cela est symbolisé par TO. Les résultats que nous présentons portent sur les instances des classes *modifiedRenault* et *geom*. Dans le tableau 1, les résultats de la version classique de $BTD_{HMIN(TD)}$ sont reportés dans la colonne temps1.

Comme nous nous y attendions, $BTD_{HMIN(HD)}$ a des performances très pauvres. Elle échoue dans la résolution de beaucoup d'instances (TO ou l'espace mémoire requis dépasse 1GB, ceci étant symbolisé par MO). Sa durée d'exécution moyenne dépasse 248s. En effet, les sous-problèmes dans une HD sont très difficiles à résoudre à cause du nombre restreint de contraintes considérées. Ce petit nombre de contraintes affaiblit la puissance des techniques de filtrage qui contribuent grandement à l'efficacité de l'énumération dans ces sous-problèmes. $BTD_{HMIN(TD)}$ se comporte largement mieux grâce à un nombre beaucoup plus grand de contraintes dans les clusters qui deviennent plus faciles à résoudre. Elle réussit à résoudre toutes les instances avec une durée moyenne de 6,67s.

Pour confirmer cette observation, nous avons essayé de mettre en lumière l'impact du nombre de contraintes dans les clusters des TD. Ainsi, nous avons considéré les méthodes $BTD_{HMIN(TD)+HMIN(HD)}$ et $BTD_{HMIN(TD)+HMAX(HD)}$. Dans $BTD_{HMIN(TD)+HMIN(HD)}$, nous avons utilisé Minfill et MCS pour calculer deux TD et nous choisissons ensuite celle qui minimise le nombre maximum de contraintes dans les clusters, tandis que dans $BTD_{HMIN(TD)+HMAX(HD)}$, nous choisissons celle qui maximise ce nombre.

Nous observons que $BTD_{HMIN(TD)+HMIN(HD)}$ et $BTD_{HMIN(TD)}$ obtiennent les mêmes résultats rassemblés dans la même colonne. $BTD_{HMIN(TD)+HMAX(HD)}$ donne les meilleurs résultats puisque les clusters très contraints sont plus facile à résoudre. Sa durée de résolution moyenne est 5,42s. En outre, les performances de $BTD_{HMIN(HD)}$ sont drastiquement améliorées si on prend en compte toutes les contraintes possibles (dans la colonne temps2) à l'image de $BTD_{HMIN(TD)}$, tandis que les bornes de complexité théorique sont préservées. Néanmoins, ces résultats restent en dessous de ceux de la méthode $BTD_{HMIN(TD)+HMAX(HD)}$. En effet, la durée moyenne de résolution de cette approche est de 6,05s.

Il faut préciser que FC, tout seul, échoue dans la résolution de près de la totalité des instances de la classe *modifiedRenault*, qui ont de bonnes propriétés topologiques (w est en moyenne inférieur à $n/10$). Alors que, ces résultats dans la classe *geom* sont meilleurs par rapport à ceux des méthodes de résolution basées décomposition car la taille de ces problèmes est plus petite et la qualité de leur propriétés topologiques est assez faible (w est en moyenne très proche de $n/2$).

En résumé, nous avons noté premièrement que calculer une TD avec une petite largeur est plus intéressant en pratique (pour la résolution) que le calcul d'une bonne HD. En effet, cette TD donnent de meilleurs résultats en pratique, de même que des bornes de complexité de qualité par rapport à la taille des clusters et de leur recouvrement minimum. En outre, ses clusters peuvent être résolus plus facilement quand ils contiennent plus de contraintes. Cela est justifié par le fait que plus un problème est contraint, plus il est facile à résoudre (ce qui peut être démontré aisément grâce à des arguments probabilistes).

5 Discussion et Conclusion

Nous avons proposé de nouvelles approches basées sur la combinaison de méthodes de décomposition de CSPs. Plus précisément, nous avons introduit deux méthodes optimales, $TC\text{-}2009_{HD}$ et $BTD\text{-}2009_{HD}$, qui exploitent la décomposition hyperarborescente et TC ou BTD pour résoudre des réseaux de contraintes. Cette approche permet d'avoir de meilleures bornes de complexité tout en héritant de l'efficacité pratique des méthodes énumératives telles que nFC2, une des techniques les plus performantes dans la résolution de CSP. Ensuite, nous avons enrichi la hiérarchie des Contraintes Traitables en mettant à jour le sommet de la hiérarchie où nous retrouvons désormais les méthodes $TC\text{-}2009_{HD}$ et $BTD\text{-}2009_{HD}$. Finalement, nous avons obtenu des résultats expérimentaux qui montrent l'intérêt pratique de notre approche.

Ce travail constitue une extension des résultats présentés dans [15]. D'une part, nous montrons, pour les méthodes de résolution basées sur la décomposition de réseaux, l'utilité d'une différenciation de la phase de décomposition et de la phase de résolution. Cette distinction nous permet de définir une plus large palette de méthodes via des hybridations entre les techniques existantes, et elle fournit également une meilleure lecture de la *Hiérarchie des Contraintes Traitables*, tout en la complétant. De plus, nous montrons expérimentalement comment il est possible d'exploiter ces méthodes hybrides. En particulier, ce travail propose des pistes nouvelles pour le choix des décompositions de réseaux et pour celui des méthodes de résolution qui permettront de les exploiter au mieux.

Nos résultats diffèrent de celui de [4] (théorème 7.28, page 231). Dans cet article, Dechter propose la résolution d'une décomposition arborescente de CSP en r^{hw} où hw est obtenu en faisant la somme du nombre de contraintes inclues dans chaque cluster et du nombre de variables du cluster qui ne sont pas couvertes par ces contraintes, puis en prenant le maximum de ces valeurs sur tous les clusters. Pour $TC\text{-}2009_{HD}$ et $BTD\text{-}2009_{HD}$, la complexité en temps est limitée à r^h (où h est la largeur hyperarborescente généralisée induite). De ce point de vue, notre résultat est meilleur.

Nos résultats diffèrent également de ceux de [5] qui démontrent de manière empirique que la borne fournie par la largeur arborescente est souvent meilleure que celle de la largeur hyperarborescente dans les réseaux de contraintes probabilistes ou déterministes. Dans le rapport technique constituant une extension de cet article, il est dit que l'approche And/Or Search Graph garantit une borne de complexité temporelle qui dépend de la largeur d'une décomposition hyperarborescente donnée. Mais, cette méthode ne considère qu'une sous classe des décompositions hyperarborescentes possibles. Cela veut dire que sa complexité est moins forte que celle de MHD, alors que celle de $TC\text{-}2009_{HD}$ et $BTD\text{-}2009_{HD}$ dépendent de la largeur d'une décomposition hyperarborescente généralisée. Donc, la complexité de ces dernières est au moins équivalente à celle de MHD.

Une poursuite naturelle de ce travail pourrait être l'étude de décompositions graphiques qui combinent l'optimisation des paramètres w (qui serait minimisé) et h (qui serait maximisé). Par ailleurs, il semble naturel d'étendre cette analyse au COP (optimisation sous contraintes) ou aux modèles graphiques probabilistes comme dans [5] (et [17]).

Références

[1] I. Adler, G. Gottlob, and M. Grohe. Hypertree width and related hypergraph invariants. *Eur. J. Comb.*, 28(8) :2167–2181, 2007.

[2] C. Bessière, P. Meseguer, E. C. Freuder, and J. Larrosa. On forward checking for non-binary constraint satisfaction. *Artificial Intelligence*, 141 :205–224, 2002.

[3] R. Dechter. *Constraint processing*. Morgan Kaufmann Publishers, 2003.

[4] R. Dechter. *Tractable Structures for Constraint Satisfaction Problems*, pages 209–244. Chapter in the *Handbook of Constraint Programming* F. Rossi, T. Walsh and P. van Beek, 2006.

[5] R. Dechter, L. Otten, and R. Marinescu. On the Practical Significance of Hypertree vs. Tree Width. In *ECAI*, pages 913–914, 2008.

[6] R. Dechter and J. Pearl. Tree-Clustering for Constraint Networks. *Artificial Intelligence*, 38 :353–366, 1989.

[7] A. Dermaku, T. Ganzow, G. Gottlob, B. MacMahan, N. Musliu, and M. Samer. Heuristic Methods for Hypertree Decompositions. In *MICAI 2008*, 2008.

[8] G. Gottlob, N. Leone, and F. Scarcello. On Tractable Queries and Constraints. In *Proceedings of the 18th Symposium on Principles of Database Systems (PODS-99)*, pages 21–32, 1999.

[9] G. Gottlob, N. Leone, and F. Scarcello. A Comparison of Structural CSP Decomposition Methods. *Artificial Intelligence*, 124 :343–282, 2000.

[10] G. Gottlob, N. Leone, and F. Scarcello. Robbers, marshals, and guards : game theoretic and logical characterizations of hypertree width. *Journal of Computer and System Sciences (JCSS)*, 66(4) :775–808, 2003.

[11] G. Gottlob and M. Samer. A backtracking-based algorithm for hypertree decomposition. *ACM Journal of Experimental Algorithmics*, 13, 2008.

[12] M. Grohe and D. Marx. Constraint solving via fractional edge covers. In *SODA*, pages 289–298, 2006.

[13] R. Haralick and G. Elliot. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14 :263–313, 1980.

[14] P. Jégou, S. N. Ndiaye, and C. Terrioux. Computing and exploiting tree-decompositions for solving constraint networks. In *CP*, pages 777–781, 2005.

[15] P. Jégou, S.N. Ndiaye, and C. Terrioux. A New Evaluation of Forward Checking and its Consequences on Efficiency of Tools for Decomposition of CSPs. In *ICTAI*, pages 486–490, 2008.

[16] P. Jégou and C. Terrioux. Hybrid backtracking bounded by tree-decomposition of constraint networks. *Artificial Intelligence*, 146 :43–75, 2003.

[17] K. Kask, R. Dechter, J. Larrosa, and A. Dechter. Unifying tree decompositions for reasoning in graphical models. *Artificial Intelligence*, 166 :165–193, 2005.

[18] N. Robertson and P.D. Seymour. Graph minors II : Algorithmic aspects of treewidth. *Algorithms*, 7 :309–322, 1986.

[19] D. Sabin and E. Freuder. Contradicting Conventional Wisdom in Constraint Satisfaction. In *Proc. of ECAI*, pages 125–129, 1994.

[20] R. Tarjan and M. Yannakakis. Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs. *SIAM Journal on Computing*, 13 (3) :566–579, 1984.

AEON : Synthèse d'Algorithmes d'Ordonnancement à partir de Modèles de Haut Niveau*

Jean-Noël Monette[1], Yves Deville[1] et Pascal Van Hentenryck[2]

[1] INGI, UCLouvain, 1348 Louvain-la-Neuve, Belgium
[2] Brown University, Box 1910, Providence, RI 02912
{jean-noel.monette,yves.deville}@uclouvain.be, pvh@cs.brown.edu

Résumé

Ce papier décrit AEON, un système dédié à la synthèse d'algorithmes d'ordonnancement à partir de modèles de haut niveau. AEON, qui est entièrement écrit en COMET, prend en entrée le modèle de haut niveau d'un problème d'ordonnancement et l'analyse pour générer un algorithme dédié et qui exploite la structure du problème. AEON offre une variété de synthétiseurs pour générer des algorithmes complets ou heuristiques. En outre, ces synthétiseurs peuvent être composés, permettant de générer naturellement des algorithmes hybrides complexes. Les premiers résultats montrent que cette approche peut être compétitive avec l'état de l'art des algorithmes de recherche.

1 Introduction

Les problèmes d'ordonnancement sont omniprésents dans l'industrie et ont été l'objet d'une important recherche depuis plusieurs dizaines d'années. Il existe, à l'heure actuelle, des algorithmes efficaces pour différentes classes de problèmes et des systèmes pour modeler et résoudre des problèmes complexes sont disponibles. Cependant, une des difficultés avec les outils existants est que l'utilisateur ne peut pas se contenter de connaître son domaine d'application, il doit aussi être expert dans les aspects algorithmiques et combinatoires. En effet, deux applications qui peuvent sembler pratiquement identiques du point de vue du modèle peuvent nécessiter des approches totalement différentes pour obtenir des solutions de bonne qualité.

Le présent travail est une première étape pour supprimer ces limitations et combler le trou entre modélisation de haut niveau et résolution efficace de problèmes d'ordonnancement. Cet article présente AEON[1], un système qui permet de transformer des modèles de haut niveau en des algorithmes efficaces en exploitant la structure du modèle. Les modèles dans AEON sont écrits avec les abstractions habituelles et leur structure est analysée pour synthétiser des algorithmes *ad hoc* pour l'application décrite. L'utilisateur décrit son modèle et se contente de choisir un synthétiseur qui va générer et exécuter un algorithme d'un paradigme particulier (par exemple, la programmation par contraintes (ppc) ou la recherche locale). Les synthétiseurs d'AEON sont compositionnels, ce qui permet de déclarer des algorithmes hybrides de façon naturelle.

Le système présente différents avantages. Du point de vue de l'utilisateur, AEON permet de se concentrer sur la description de son application à un niveau élevé d'abstraction, le déchargeant de s'occuper des aspects algorithmiques. De plus, comme les modèles révèlent la structure des applications, les synthétiseurs d'AEON sont capables d'exploiter toute la richesse de la recherche en ordonnancement pour dériver des algorithmes efficaces. Finalement, grâce à la séparation nette entre le modèle et la résolution, AEON permet d'appliquer des paradigmes de recherche différents et de développer des hybridations, dont le potentiel a été démontré pour de nombreuses applications. Au niveau de l'implémentation, AEON présente aussi des innovations. Premièrement, l'analyse du modèle est extensible et permet d'ajouter des nouvelles classes de problèmes en les décrivant selon un format XML standard. Deuxièmement, de nouveaux synthétiseurs peuvent être ajoutés simplement et compositionnellement. Finalement, plusieurs abstractions simplifient l'écrit-

*Traduction française d'un article présenté à ICS09 [6]

[1] Aeon est un nom alternatif pour le dieu du temps Chronos. Cela signifie éternité.

ure de synthétiseurs. En particulier, AEON fournit des *vues du modèles*, qui permettent d'accéder à un modèle général et à sa solution à travers une interface spécialisée pour un problème particulier.

Ce papier étend et généralise la recherche commencée dans [11] qui montrait comment synthétiser des algorithmes de recherche locale à partir de modèles COMET. Les synthétiseurs proposés ici s'appliquent plus particulièrement à l'ordonnancement de tâches, considèrent différents paradigmes de recherche (algorithmes gloutons, recherche locale, ppc) et sont extensibles et compositionnels. Le style des modèles utilisé est similaire à ceux de ILOG Scheduler, OPL, COMET et d'autres systèmes d'ordonnancement basés sur les contraintes. ILOG Concert propose aussi une couche de modélisation qui peut être utilisée par des solveurs différents, mais il n'y a pas de tentative de synthétiser des algorithmes de recherche. Il est par ailleurs utile de contraster notre travail avec des travaux récents en ppc qui visent la conception de procédures de recherche par défaut. Parmi d'autres, [4] décrit une recherche par voisinage large auto-adaptive et [8] présente l'utilisation d'impacts pour guider la recherche. Leur but est de proposer une procédure de recherche robuste sur une large gamme de problèmes. Au contraire, notre objectif est d'exploiter la structure du modèle pour dériver des algorithmes spécifiques. Ces deux approches sont orthogonales car il faut aussi des procédures robustes pour différents types de problèmes. Cependant, révéler et exploiter la structure d'un modèle est une des principales contributions de la ppc et les algorithmes de recherche peuvent bénéficier énormément d'une analyse de la structure.

Le reste de cet article présente une vue globale des différentes parties du système. La Section 2 couvre l'utilisation d'AEON et les abstractions et synthétiseurs disponibles. Dans la Section 3, l'architecture est présentée et certaines caractéristiques du système sont mise en valeur. Ensuite, la Section 4 montre comment le système peut être étendu avec d'autres familles de problèmes ou d'algorithmes. La Section 5 présente et analyse des résultats expérimentaux.

2 Modélisation et Résolution de Problèmes d'Ordonnancement

La Figure 1 présente un modèle AEON pour la problème du Job-Shop (JSP) et un synthétiseur. L'initialisation des données aux lignes 1–6 n'est pas montré. Le modèle lui-même est donné aux lignes 8–16. D'abord un objet est créé pour le problème. Ensuite, les objets qui peuplent ce problème sont créés (lignes 9–11), activités, jobs et machines. Après les contraintes sont posées : demandes des machines (lignes 12-13), ordre

```
1 range jobs = 1..nbjobs;
2 range machines = 0..nbmachines-1;
3 range tasks = 1..nbjobs*nbmachines;
4 int proc[tasks];
5 int mach[tasks];
6 int job[jobs,machines];
7
8 Schedule<Mod> s();
9 Job<Mod> J[i in jobs](s,IntToString(i));
10 Machine<Mod> M[i in machines](s,IntToString(i));
11 Activity<Mod> A[i in tasks](s,proc[i],
   IntToString(i));
12 forall(i in tasks)
13    A[i].requires(M[mach[i]]);
14 forall(i in jobs)
15    J[i].containsInSequence(all(j in machines)A[
      job[i,j]]);
16 s.minimizeObj(makespanOf(s));
17
18 GreedyTabuSynthesizer synth();
19 Solution<Mod> sol = synth.solve(s);
20 sol.printSolution();
```

FIG. 1 – Un modèle pour le Job-Shop et un synthétiseur.

dans les jobs (lignes 14-15). Finalement, l'objectif à la ligne 16 minimise la fin des activités (makespan).

Les trois dernières lignes concernent la résolution. La ligne 18 définit le synthétiseur qui crée, dans ce cas, une recherche gloutonne suivie d'une recherche tabou. Il est facile de modifier le synthétiseur : il suffit de remplacer `GreedyTabuSynthesizer` par `CPSynthesizer` pour obtenir une recherche par ppc. La ligne 19 applique le synthétiseur au modèle. Cela entraîne l'analyse et la classification du modèle, la génération des variables contraintes et objectifs appropriés aux solvers et l'exécution d'un algorithme de recherche dédicacé au modèle. Le synthétiseur produit une solution qui peut être utilisée par la suite. Par exemple, la ligne 20 imprime la solution.

Il est clair sur la Figure 1 que la modélisation et la résolution sont clairement séparées. AEON offre un riche ensemble d'abstractions (classes, méthodes, fonctions) pour modéliser une large gamme de problèmes d'ordonnancement. Le reste de cette section passe en revue les abstractions disponibles à l'heure actuelle.

Les classes de modélisation se terminent par "<Mod>" pour dénoter qu'elles servent pour le modèle[2]. A l'intérieur du système, d'autres classes sont post-fixées avec "<CP>" ou "<LS>" et servent aux

[2]Malgré la syntaxe similaire au C++, il ne s'agit pas de classes template.

TAB. 1 – Résumé des classes disponibles pour la modélisation.

Description	Classes
Problème	`Schedule`
Activités	`Activity`
	`MultiModeActivity`
Jobs	`Job`
Ressources	`Resource`
	`Machine`
	`Reservoir`
	`StateResource`
Objectifs	`ScheduleObjective`
	`TaskObjective`
	`CompletionTime`
	`Lateness`
	`Tardiness`
	`Earliness`
	`UnitCost`
	`PiecewiseLinearFunction`
	`AbsenceCost`
	`AlternativeCost`
	`ModifObjective`
	`MultObjective`
	`ShiftObjective`
	`AgregObjective`
	`SumObjective`
	`MaxObjective`

algorithmes de recherche. Par exemple, le but de la classe `Activity<Mod>` est complètement différent de celui de la classe `Activity<CP>`. Bien qu'elles soient associées au même concept (une activité), la première propose des méthodes pour faire l'analyse du modèle, tandis que la seconde encapsule les variables de ppc qui représentent le moment de début et de fin d'une activité et une contrainte qui les relie. Pour simplifier la lecture, le postfixe "`<Mod>`" est omis dans cette section quand il est clair qu'on se réfère aux classes de modélisation.

La Table 1 présente les classes de modélisation disponibles actuellement dans AEON et dont voici les explications. La classe centrale est `Schedule` (i.e. `Schedule<Mod>`). Elle est passée en paramètre à la création de tous les autres objets et est responsable de la cohérence interne du modèle. Pour représenter les activités, il y a deux classes. `Activity` et `MultiModeActivity` représentent respectivement des activités avec un ou plusieurs modes. A sa création, une activité reçoit en entrée un schedule, une durée et un nom. La durée est soit fixée, soit définie par des bornes supérieure et inférieure. Une `MultiModeActivity` est initialisée avec un `Schedule`, le nombre de modes et un nom. La durée de chacun des modes est donnée séparément pour chaque mode. Les méthodes sur les activités permettent de spécifier l'appartenance à un `Job`, les besoins en ressources et les précédences entre activités. Les besoins dépendent des modes mais les autres contraintes sont communes à tous les modes. Les contraintes de précédence peuvent faire intervenir le début et la fin des activités et des jobs, ainsi que des délais. La classe `Job` représente des groupes d'activités. Les activités d'un job ne sont pas nécessairement ordonnées mais elles ne peuvent être exécutées en même temps. Les jobs partagent certaines caractéristiques des activités : Ils peuvent eux-mêmes être regroupés dans d'autres jobs, et leurs débuts et fins peuvent être contraints avec des précédences. Finalement, les activités et les jobs peuvent être définis comme optionnels, auquel cas ils ne doivent pas obligatoirement être exécutés.

Les ressources sont représentées par quatre classes, en fonction du type de ressource considéré. La classe `Machine` représente des ressources disjonctives. Deux activités qui ont besoin d'une même machine ne peuvent être exécutées en même temps. La classe `Resource` représente les ressources renouvelables. A tout moment, la somme des besoins des activités qui sont exécutées ne peut dépasser la capacité de la ressource. Au contraire, la classe `Reservoir` représente des ressources non-renouvelables et dont la capacité est diminuée à la fin de l'exécution de chaque activité. Une capacité minimum peut être définie pour les classes `Resource` et `Reservoir`. Pour ces deux classes et la classe `Machine`, il est aussi possible de définir des pauses (périodiques), i.e. des moments d'indisponibilité. Le dernier type de ressources est la classe `StateResource` qui représente un état du monde. Une telle ressource ne peut être que dans un état à la fois. Deux activités qui ont besoin d'états différents ne peuvent s'exécuter en même temps. Pour tous les types de ressources, il est possible de définir des temps et des coûts de setup qui dépendent de l'ordonnancement. L'ensemble des besoins d'une activité (ou d'une activité à plusieurs modes) a la forme d'un arbre dont les noeuds internes sont des conjonctions ou des disjonctions de besoins plus simples. Les noeuds externes sont les besoins de base : besoin d'une machine, une quantité de ressource demandée ou fournie, une quantité consommée ou produite dans un réservoir, un état particulier d'une ressource à états.

Les fonctions objectif sont des sous-classes de `ScheduleObjective`. Les sous-classes sont des fonctions simples ou composées. Les fonctions composées sont la somme, le minimum, le maximum d'autres fonctions et la multiplication par une constante. Les fonctions simples sont les classiques date de fin, retard, avance et de façon plus générale les fonctions linéaires

TAB. 2 – Résumé des classes disponibles pour la résolution.

Description	Classes
Synthétiseurs	`ScheduleSynthesizer`
	`CPSynthesizer`
	`TSSynthesizer`
	`SASynthesizer`
	`GreedySynthesizer`
	`SequenceSynthesizer`
	`ScheduleAnimator`
Solutions	`Solution`

par morceaux basées sur la date de fin des activités et des jobs. Les fonctions simples sont aussi le coût associé au mode des activités multimodales, à l'absence d'une activité optionnelle ou au setup des ressources. La fonction objectif globale est passée au `Schedule` par une méthode qui spécifie s'il s'agit d'une minimisation ou d'une maximisation. La fonction `makespanOf` à la Figure 1 est un raccourci pour le makespan, maximum des dates de fin, qui est un objectif commun et important.

Cet ensemble d'abstractions permet de modéliser des problèmes aussi variés que les classiques problèmes d'"atelier" (Job Shop, Open Shop, Flexible Shop, Flow Shop, Group Shop, Cumulative Shop, Just-In-Time Job Shop), des variations de l'ordonnancement de projet avec des contraintes de ressources (RCPSP, MRCPSP, RCPSP/max, MRCPSP/max) ([3]), le problème du trolley ([12]) and les classes NCOS et NCGS de MaScLib ([7]). Cela représente des problèmes avec différents types d'objectifs et différentes propriétés (disjonctif ou cumulatif, un ou plusieurs modes).

Bien que les abstractions de modélisation permettent de représenter un grand nombre de problèmes, l'ensemble des problèmes qui peut être résolu dépend de la recherche qui peut être synthétisée. La Table 2 présente les classes de synthèse qui sont disponibles dans AEON actuellement. Pour le moment, il y a trois solveurs sous-jacents : la ppc, la recherche locale (tabou et recuit simulé) et la recherche gloutonne.[3] Leurs possibilités définissent les possibilités de tout le système. Des solveurs plus complexes peuvent être synthétisés à partir de ceux de base. En particulier, il est possible de faire des solveurs hybrides et animés. Par exemple, un solveur animé emballe un solveur sous-jacent dans un environnement visuel qui montre la succession des solutions trouvées. Des solveurs hybrides peuvent être de simples séquences de solveurs ou peuvent suivre des schémas de décomposition plus compliqués. Les synthétiseur acceptent

[3]Dans le futur, nous allons aussi intégrer des solutions basées sur la programmation mathématique.

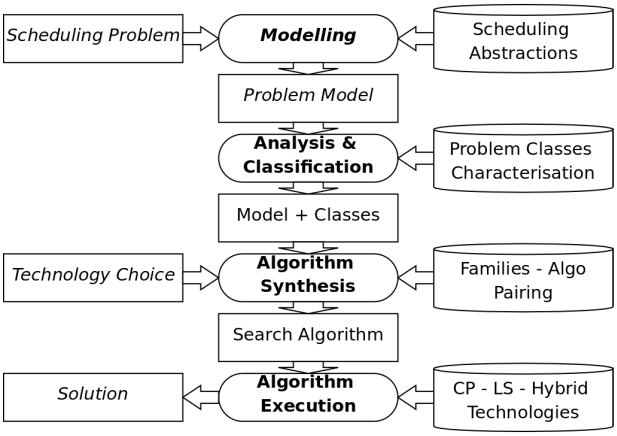


FIG. 2 – Vue d'ensemble d'AEON. Les rectangles arrondis représentent les actions pour résoudre le problème. Les rectangles et conteneurs représentent leurs entrées et sorties. Les conteneurs sur la droite sont fournis par le système, les rectangles sur la gauche sont les entrées de, et les sorties vers l'utilisateur et les rectangles au milieu sont des produits intermédiaires. Le texte en italique représente les endroits d'implication de l'utilisateur.

aussi des paramètres, comme par exemple une limite temporelle sur la recherche.

3 Architecture

La Figure 2 présente une vue d'ensemble d'AEON centrée sur la résolution d'un problème. Les rectangles arrondis sont les étapes successives vers une solution. Seule la première, la modélisation, fait participer l'utilisateur. Ensuite, le modèle est analysé et catégorisé dans des classes de problèmes. Un algorithme est alors synthétisé et exécuté pour produire une solution au problème. Les conteneurs sur la droite de la Figure 2 représentent ce qui est fourni par le système à chaque étape. Le reste de cette section explique plus en détails l'exécution de chaque étape. La Section 4 décrit comment il est possible d'agrandir les conteneurs.

3.1 Modélisation

La Section 2 a présenté la modélisation du point de vue de l'utilisateur. A l'intérieur, quand le modèle est exécuté, une représentation interne du problème est construite. La majorité de l'information est enregistrée dans les objets de modélisation qui ont été présentés. Par exemple, la classe `Activity` contient un attribut retenant si la préemption est autorisée ou pas. En outre, l'objet `Schedule` garde une référence vers tous les objets qui ont été créés. L'information est

enregistrée grâce à des structures de graphes : les relations de précédences dans un graphe dirigé, les fonctions objectif et les besoins en ressources dans des arbres plantés. Les contraintes de précédence sont des arcs étiquetés d'un graphe dont les noeuds représentent le début et la fin des activités, des jobs et du schedule. En plus des arcs ajoutés explicitement par l'utilisateur, il y a des arcs qui relient le début d'un job avec le début des activités contenues et la fin d'un job avec la fin des activités contenues. Il y a aussi des arcs qui assurent que toutes les activités et tous les jobs sont exécutés entre le début et la fin du schedule. Les arbres plantés représentant les fonctions objectifs et les besoins en ressources sont nécessaires pour représenter la combinaison de fonctions ou besoins simples. Les combinaisons sont des sommes, des produits, des minimums et des maximums pour les objectifs. Il s'agit de conjonctions et disjonctions pour les besoins.

3.2 Analyse et Classification

Le but de la seconde étape est de catégoriser le modèle dans une des classes prédéfinies. Cette classification est basée sur les caractéristiques du problème. Chaque classe de problèmes est définie par une combinaison de paires (caractéristique, valeur). La Table 3 présente un sous-ensemble des caractéristiques considérées. La dernière colonne spécifie les valeurs pour une classe bien connue. Un tiret signifie que la valeur peut être n'importe quoi. Cette table représente une version simplifiée de la définition des classes. En fait, il ne s'agit pas d'une simple conjonction de paires mais plutôt d'une formule booléenne, avec des négations, des disjonctions et des conjonctions de formules plus simples. Les atomes correspondent aux paires. Leur valeur de vérité est déterminée par analyse du modèle. Si la valeur renvoyée par l'analyse est égale à celle attendue, la formule atomique reçoit la valeur *Vrai*. Un modèle appartient à une classe de problèmes si l'évaluation de la formule qui définit la classe est *Vrai*.

De plus, des sous-formules récurrentes sont définies comme des caractéristiques de haut niveau, ou comme des modèles plus généraux dont les autres modèles peuvent hériter. Par exemple, le JSP avec Makespan est un cas spécial du JSP auquel on ajoute la caractéristique "avec Makespan". Le JSP est à son tour un cas de problème disjonctif. La hiérarchie des catégories forme un graphe dirigé acyclique (DAG). Cela signifie qu'un problème catégorisé dans une classe est aussi membre de toutes les classes dont elle hérite. Le résultat de la classification est donc une séquence de classes, plutôt qu'une seule classe. Cette séquence représente un ordre total des classes du problème compatible avec le DAG des classes. Cela signifie que, si une classe hérite d'une autre, elle doit apparaître avant son parent dans la séquence. Par contre l'ordre de classes qui ne sont pas apparentées est fixé arbitrairement.

Tab. 3 – Liste partielle des caractéristiques. La première colonne donne la caractéristique, la seconde définit le type de valeur. La troisième illustre les valeurs possibles pour le problème du Job-Shop (JSP).

Caractéristique	Type	JSP
Unit Processing Time	boolean	–
Fixed Processing Time	boolean	true
Preemption Allowed	enum	never
Common Release Dates	boolean	true
Common Deadlines	boolean	–
Deadlines Exist	boolean	false
Form of the Precedence Graph	enum	chains
Delay between Activities	boolean	false
No wait between Activities	boolean	false
Jobs inside Jobs	boolean	false
Number Of State Resources	integer	0
Maximum Capacity	integer	1
All Capacities are Equal	boolean	true
Reservoir Consumption	boolean	false
Reservoir Production	boolean	false
Setup Times	boolean	false
Disjunctive Requirements	boolean	false
All Activities in Jobs	boolean	true
Nb of Multi-Mode Activities	integer	0
Sum Of Requirements	integer	1
Objective Type	enum	minimize
Objective Form	enum	maximum
Objective Components	enum	end time
Objective Scope	enum	all activ.
All Due-Dates are equal	enum	–

L'analyse des caractéristiques en soi est accomplie par un ensemble de fonctions qui récupèrent l'information à partir de la représentation interne présentée plus haut. Avant l'analyse, une étape de normalisation est exécutée sur cette représentation. En particulier, le graphe de précédences est simplifié pour retirer les contraintes inutiles (réduction transitive). Les arbres pour les besoins et les objectifs sont aussi simplifiés. Par exemple, une somme de sommes est remplacée par une seule somme. Pour finir, les objets inutiles (machines inutilisées, jobs vides, par exemple) sont marqués comme tels.

Pour être utile, l'analyse doit être robuste aux variations de modélisation. Aeon compile les modèles dans une forme normalisée et l'analyse est accomplie sur la forme normalisée. Par exemple, le code de la Figure 3 montre une formulation alternative pour le JSP. Il y a plusieurs différences (activités multimodale, réservoirs, pas de jobs, fonction objectif explicite) par rapport au code de la Figure 1. Cependant, aeon le caté-

```
1 range machines;
2 range tasks;
3 int proc[tasks];
4 int mach[tasks];
5
6 Schedule<Mod> s();
7 Reservoir<Mod> M[i in machines](s,0,5,5,
  IntToString(i));
8 MultiModeActivity<Mod> A[i in tasks](s,1,
  IntToString(i));
9 forall(i in tasks) {
10   A[i].setProcTime(1,proc[i],proc[i]);
11   A[i].requires(1,M[mach[i]],3);
12 }
13 forall(i in tasks:i%nbmachines!=0)
14   A[i].precedes(A[i+1]);
15 s.minimizeObj(maxOf(all(i in tasks)
  completionTimeOf(A[i])));
16
17 GreedyTabuSynthesizer synth();
18 Solution<Mod> sol = synth.solve(s);
19 sol.printSolution();
```

FIG. 3 – Modèle Alternatif pour le Problème du Job-Shop

gorise correctement comme étant un JSP, ce qui est très désirable en pratique. En effet, c'est la sémantique du modèle qui est importante, pas la syntaxe.

3.3 Synthèse d'Algorithmes

Les classes responsables de la synthèse sont `ScheduleSynthesizer` et ses sous-classes (voir Table 2). Comme réfléchi sur la Figure 2, l'entrée de la synthèse est composée de trois parties : le modèle de l'utilisateur, sa classification et un choix fait par l'utilisateur pour un paradigme de recherche en particulier. La sous-classe de `ScheduleSynthesizer` choisie définit le paradigme de recherche (par exemple, la ppc pour `CPSynthesizer`) et la méthode `solve` prend le modèle en argument.

A partir du résultat de la classification, le synthétiseur choisit la stratégie de résolution appropriée. Une stratégie est un algorithme de recherche spécifique à une classe de problèmes et qui va être instantié pour une instance particulière. Chaque synthétiseur associe une stratégie à chaque classe de problèmes. Par exemple, la classe `TSSynthesizer` associe la recherche Tabou de [2] à la classe Job-Shop avec Makespan. Chaque synthétiseur peut ne pas définir une stratégie pour toutes les classes de problèmes mais il est possible qu'il définisse une stratégie pour un problème plus général. Comme le résultat de la classification est un séquence de classes de problèmes, le synthétiseur cherche une stratégie pour la première classe. S'il n'y en a pas, il cherche pour la classe suivante. La séquence est visitée tant qu'il n'y a pas de stratégie qui correspond à une classe. Dans le pire des cas, le problème est reconnu comme un "problème d'ordonnancement général" pour lequel il y a une recherche par défaut de base.

Une fois la stratégie choisie, elle doit être instanciée pour le problème à résoudre. Le synthétiseur délègue ce travail à une sous-classe de la classe `Strategy`. Il y a à peu près une telle sous-classe pour chaque paire formée d'une classe de problème et d'un paradigme de recherche. Chaque sous-classe de `Strategy` est responsable de la mise en place et de l'exécution d'un algorithme pour le problème à résoudre. La difficulté est que, bien que la classe du problème soit connue, il peut être difficile de trouver l'information nécessaire à l'instanciation de la recherche. Pour faciliter cette étape, AEON fournit un ensemble de classes appelées vues. Les vues sont utilisées pour présenter le schedule et ses composants de manière unifiée, quelle que soit la façon dont ils ont été introduits. Différentes vues correspondent à différentes conceptions du problème. La vue la plus générale (`ScheduleView`) est une façon générique d'accéder à l'information, tandis que des vues spécifiques donnent un accès direct à un sous-ensemble de l'information utile pour certaines classes de problèmes. Par exemple, la vue `JobShopView` donne de l'information pour les JSPs. Elle fournit la même interface, quelle que soit la façon dont le problème a été modélisé par l'utilisateur (comme à la Figure 1 ou comme à la Figure 3).

3.4 Exécution de l'Algorithme

L'algorithme effectivement exécuté est différent pour chaque stratégie. Cependant, ils ont en commun qu'une solution est renvoyée. Les objets de la classes `Solution` assignent une valeur à chaque variable de décision du problème. Cette assignation est exprimée en fonction des objets du modèle. Par exemple, la méthode `getStartingTime( Activity<Mod> act)` renvoie le moment de départ d'une activité. En plus des temps de départ, les autre variables de décisions des activités sont la date de fin, l'ensemble des ressources effectivement utilisées, le mode (si il y en a plusieurs) et la présence ou l'absence (si elles sont optionnelles). La solution enregistre aussi la valeur de la fonction objectif associée. Le principal bénéfice des objets de solution est que le modèle reste indépendant. Il peut donc avoir plusieurs solutions qui peuvent être comparées. De plus les solutions peuvent servir de moyen de communication entre des stratégies qui coopèrent. Elles peuvent ainsi être utilisées

```
1 Solution<Mod> solve(Schedule<Mod> sched){
2   JobShopView view(sched);
3   range Activities = view.getActivities();
4   range Jobs = view.getJobs();
5   range Machines = view.getMachines();
6   int[] duration = all(i in Activities)
7                      view.getProcessingTime(i);
8   int[] machine = all(i in Activities)
9                            view.getMachine(i);
10  int[][] jobAct = all(j in Jobs)
11             view.getOrderedActivitiesOfJob(j);
12  JobshopAlgorithm ls(LocalSolver(),Activities,
13     Jobs, Machines, duration, machine, jobAct);
14  ls.solve();
15  SolutionView sol(view);
16  ls.saveSolution(sol);
17  return sol.getModelSolution();
18 }
```

FIG. 4 – Résoudre un Problème de Job-Shop

comme assignation de départ, pour fournir une borne sur l'objectif, ou pour guider des heuristiques.

Les solutions sont exprimées en fonction du modèle mais les stratégies travaillent sur des vues. Elles ont besoin d'une classe `SolutionView` pour exprimer la solution à partir de la vue. Un objet `SolutionView` est créé à partir d'une vue et les valeurs pour les variables de décision sont données dans les termes de la vue. La solution du modèle sous-jacente peut ensuite être récupérée à partir de sa vue. La Figure 4 présente le corps de la méthode `solve` de la class `DellAmico`. On y retrouve les classes `JobShopView` et `SolutionView`. Les lignes 2-11 montrent la création et l'utilisation de la vue pour les problèmes de Job-Shop. La recherche effective est déléguée à une autre classe appelée `JobshopAlgorithm` (lignes 12-14). La ligne 15 crée la vue pour la solution à partir de la vue du problème. Cette vue est ensuite remplie (ligne 16) et la solution est renvoyée à la ligne 17.

4 Ajouter des Classes de Problèmes et des Stratégies

Comme le principal objectif de ce travail est de simplifier l'utilisation d'algorithmes d'ordonnancement, il est aussi important de fournir des moyens simples d'étendre le système. En particulier, l'architecture d'AEON permet d'ajouter facilement des classes de problèmes, des synthétiseurs et des stratégies. L'extension des abstractions de modélisation n'est pas couvert car cela nécessite une plus grande modification du système. De nouvelles classes de problèmes peuvent être

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <!DOCTYPE Model SYSTEM "models.dtd">
3 <Model ID="JobShopWithMakespanWithTwoJobs">
4   <Constraints>
5     <IsA Name="Makespan"/>
6     <IsA Name="JobShop"/>
7     <Constraint Name="nbJobs" Value="2"/>
8   </Constraints>
9 </Model>
```

FIG. 5 – Définition XML du Job-Shop avec deux jobs.

définies dans des fichiers XML. Les synthétiseurs et les stratégies sont définis en étendant des classes existantes.

4.1 Ajout de Classes de Problèmes

Toutes les classes de problèmes et les caractéristiques de haut niveau sont définies dans des fichiers XML. Chaque classe est définie par un nom unique et une structure de contraintes que le problème doit respecter. Cette structure est récursivement composée des éléments suivants :

- SimpleConstraint : La caractéristique doit prendre une certaine valeur.
- And : Toutes les contraintes doivent être respectées.
- Or : Au moins une contrainte doit être respectée.
- Not : La contrainte ne peut pas être respectée.
- IsA : Les contraintes d'un autre modèle doivent aussi être respectées.

L'élément racine est appelé "Constraints" et il correspond à un "And". Pour ajouter un nouvelle classe, il faut écrire un fichier XML qui définit les contraintes à satisfaire. Il est facile de réutiliser les modèles existants grâce à l'héritage définit par l'élément "IsA". Par exemple, la Figure 5 montre un fichier pour le cas particulier du JSP avec exactement deux jobs qui peut être résolu en temps polynomial ([1]). Il s'agit d'une conjonction des contraintes que ce soit un Job-Shop, que l'objectif soit la minimisation du makespan et que le nombre de jobs soit deux.

4.2 Ajout de Stratégies

Une nouvelle stratégie est créée en étendant la classe `Strategy`, ce qui demande d'écrire deux méthodes : `solve(Schedule<Mod> s)` et `solve(Schedule <Mod> sched, Solution<Mod> initSol)`. La première méthode implémente la résolution d'un problème depuis le début et la seconde la résolution d'un problème en utilisant une solution existante.

```
1 class MySynthesizer extends TSSynthesizer{
2   MySynthesizer():TSSynthesizer(){
3     registerStrategy("JobShopWMakespanW2Jobs",
4                 new AkersAndFriedmanAlgorithm());
5   }
6 }
```

FIG. 6 – Ajouter une stratégie à TSSynthesizer

La solution initiale peut être laissée de côté, par exemple dans le cas d'une recherche gloutonne. Le corps de ces méthodes doit utiliser les vues. Cela est illustré sur la Figure 4 qui montre l'implémentation de la première méthode. La seconde est similaire. La seule modification est le remplacement de la ligne 14 par l'instruction `ls.solve(new SolutionView(view,initSol))` où une vue de la solution initiale est transmise à l'algorithme de recherche.

Une stratégie nouvellement créée doit être liée à une classe de problèmes au moyen d'un synthétiseur. Cet appariement se fait par la méthode `registerStrategy(string name, Strategy strategy)` définie dans la classe `Synthesizer`. Cette méthode associe une classe (définie par son nom) à une stratégie. Si une autre stratégie était déjà associée à la classe, la nouvelle remplace l'ancienne. Cette méthode est typiquement appelée dans le constructeur d'une nouvelle classe de synthétiseur. La Figure 6 montre un tel cas, ou un nouveau synthétiseur est définit comme une sous-classe de `TSSynthesizer`. Cela signifie qu'un JSP avec deux jobs va être résolu avec l'algorithme polynomial *ad hoc* et que les autres problèmes sont résolus avec une recherche Tabou.

Sur cet exemple, il est aussi clair que le choix de l'utilisateur pour un paradigme de recherche particulier (à la Figure 2) peut aussi être enlevé, permettant une recherche entièrement en boîte noire. Il suffit de créer un synthétiseur par défaut qui associe la meilleure stratégie à chaque classe de problèmes. Cependant, la meilleure stratégie n'est pas spécialement unique, même pour une sous-classe. Cela peut dépendre de contraintes temporelles, du besoin d'avoir des bornes supérieures et inférieures, du besoin d'optimalité et des caractéristiques individuelles de l'instance. Fournir plusieurs synthétiseurs augmente donc la flexibilité et l'efficacité du système.

4.3 Création de Nouvelles Stratégies de Façon Compositionnelle

De nouvelles stratégies peuvent aussi être construites à partir d'autres plus simples. L'architecture

```
1 class TS_CPJSP extends Strategy{
2   Strategy _s1;
3   Strategy _s2;
4   TS_CPJSP():Strategy(){
5     _s1 = new DellAmico();
6     _s2 = new CPJobShop();
7   }
8   Solution<Mod> solve(Schedule<Mod> s){
9     return _s2.solve(s,_s1.solve(s));
10  }
11  Solution<Mod> solve(Schedule<Mod> s,Solution<
    Mod> initSol){
12    return _s2.solve(s,_s1.solve(s, initSol));
13  }
14 }
15 class TS_CPSynthesizer extends
   ScheduleSynthesizer{
16  ScheduleSynthesizer _s1;
17  ScheduleSynthesizer _s2;
18  TS_CPSynthesizer():ScheduleSynthesizer(){
19    _s1 = new TSSynthesizer();
20    _s2 = new CPSynthesizer();
21  }
22  Solution<Mod> solve(Schedule<Mod> s){
23    string[] models = classify(s);
24    return _s2.solve(s,models,_s1.solve(s,
      models));
25  }
26 }
```

FIG. 7 – Deux implémentations pour une stratégie TS+CP

d'AEON permet de fabriquer des recherches composites par spécialisation ou par composition. La première possibilité est de créer une nouvelle stratégie pour une classe spécifique comme montré dans la sous-section précédente. A un niveau plus général, un synthétiseur peut créer systématiquement des stratégies composées à partir d'autres synthétiseurs. La Figure 7 présente les deux possibilités pour une composition simple : une recherche Tabou suivie d'une recherche en ppc. Les lignes 1-14 illustrent une stratégie composée pour le JSP et les lignes 15-26 montrent le code d'un synthétiseur enchaînant TS et CP. Les méthodes `classify` et `solve` avec plusieurs arguments sont définies dans la classe `ScheduleSynthesizer` et représentent les différentes étapes qui sont du ressort du synthétiseur : la classification et la résolution (avec ou sans solution initiale). Il est intéressant de voir comment le code du synthétiseur composé ressemble au code de la stratégie composée.

5 Expérimentations

Le but de cette section est de montrer que la générécité du système est compatible avec des résolutions effectives et efficaces de problèmes d'ordonnancement. Pour évaluer cela, nous avons choisi d'effectuer des expérimentations sur quelques jeux de test classiques, le problème du Job-Shop avec minimisation du makespan (JSP), le problème de l'Open-Shop avec minimisation du makespan (OSP) et le problème du Job-Shop avec minimisation du retard pondéré total (JSPWT). Pour chaque jeu de test, trois algorithmes synthétisés vont être considérés : une recherche locale (LS), une approche par ppc (CP) et un composé où une recherche Tabou donne une borne supérieure à la partie CP (LS+CP). Ces algorithmes vont être comparés avec l'implémentation en COMET [10] de respectivement la recherche Tabou de [2] pour le JSP, la recherche Tabou de [5] pour l'OSP et un algorithme de Metropolis [9] pour le JSPTW.

Les algorithmes LS sont les homologues des algorithmes originaux et ont les mêmes limites : 12.000 itérations pour le JSP et l'OSP et 600.000 pour le JSPTW. La recherche CP est limitée en temps à $max(300, 3 * \#\text{activités})$ secondes, c'est-à-dire 25 minutes pour les plus grandes instances.

Pour les algorithmes de recherche locale, vingts exécutions sont faites pour chaque instance. Ceux avec CP n'ont été exécutés qu'une fois car ils sont bien moins variables. Toutes les exécutions ont été réalisées sur un Intel Core 2 Duo, 1.66Ghz avec 1 Go de RAM.

La Table 4 présente un résumé des résultats pour le jeu de test. Des résultats plus détaillés se trouvent en ligne[4]. Pour chaque algorithme, l'erreur relative moyenne (MRE) est donnée. Le MRE est égal à $100 * (UB - LB)/LB$ où UB est la valeur moyenne de la solution trouvée par l'algorithme et LB est la meilleure borne inférieure connue pour chaque instance (et récupérée dans [14, 13, 10, 4]).

Pour illustrer une autre recherche hybride, nous avons aussi généré un recherche par voisinage large (LNS) pour l'OSP. Cette recherche est particulièrement efficace et a résolu toutes les instances sauf une en moins de deux minutes.

Cette table montre qu'il n'y a pas de différence significative entre une recherche générée par AEON et une recherche écrite à part. Bien sûr, l'approche CP n'est pas toujours utilisable pour les plus grands problèmes mais ce n'est pas une particularité d'AEON. Au contraire, l'utilisation de CP en conjonction avec une recherche locale permet de prouver l'optumalité de solutions trouvées heuristiquement. Concernant les temps d'exécution, l'approche CP n'est compétitive pour les problèmes plus grands mais la recherche locale est comparable à l'implémentation en COMET des algorithmes de référence.

[4] http ://becool.info.ucl.ac.be/aeon

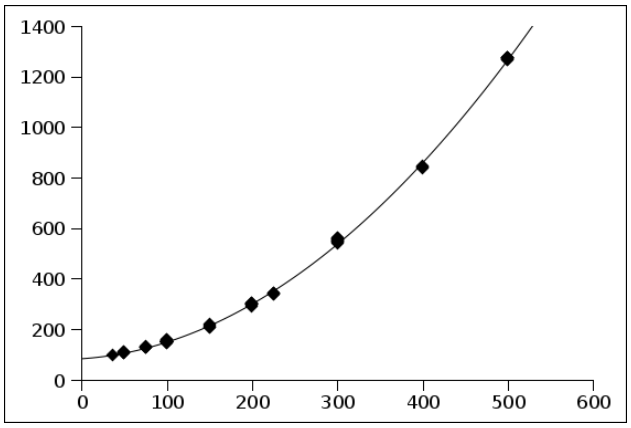


FIG. 8 – Temps (en millisecondes) pour analyser un problème et générer la recherche, en fonction du nombre d'activités.

Le coût induit par l'utilisation d'AEON est illustré à la Figure 8. Le temps utilisé pour la mise en place d'AEON et les opérations d'analyse, de classification et de génération est affiché en fonction du nombre d'activités du problème. La courbe suit une lente progression quadratique. Le temps d'analyse est de moins de 1,5 secondes même pour des problèmes à 500 activités.

6 Conclusion

Ce papier présente AEON, un système pour modéliser et résoudre des problèmes d'ordonnancement. Étant donné un modèle décrit selon un langage de modélisation de haut niveau, AEON reconnaît et classe sa structure et synthétise un algorithme de recherche approprié. L'algorithme synthétisé appartient à un paradigme particulier, tel que la recherche locale ou la ppc. L'approche permet d'exploiter la structure des modèles pour dériver des algorithmes dédicacés à des classes de problèmes.

AEON a certains traits fondamentaux : en premier, la classification du modèle ne dépend pas de la syntaxe ou des choix de modélisation. Les modèles sont transformés dans une forme normalisée sur laquelle l'analyse est effectuée, ce qui permet d'augmenter la robustesse de la modélisation. Deuxièmement, AEON est ouvert et extensible : de nouvelles classes de problèmes sont spécifiée en XML et des stratégies de recherche peuvent être ajoutées pour toutes les classes de problèmes. En outre, de nouveaux synthétiseurs peuvent être fabriqués à partir d'autres existants de façon simple.

TAB. 4 – Erreur relative moyenne (MRE) et temps d'exécution (en secondes) pour quatre algorithmes. Ref. désigne les algorithmes de référence, LS pour la recherche locale dans AEON, CP pour la ppc dans AEON et LS+CP pour un composé de LS et CP. Pour CP et LS+CP, le nombre entre parenthèses est le nombre d'instances pour lesquelles la recherche s'est finie et pour lesquelles le temps est compté. Pour l'OSP, la colonne CP présente deux valeurs. La deuxième est une recherche par voisinage large (LNS) qui est aussi générée par AEON.

Problème	#Inst.	MRE moyen				Temps moyen pour la meilleure solution			
		Ref.	LS	CP/LNS	LS+CP	Ref.	LS	CP/LNS	LS+CP
JSP	78	2.08	2.09	54.40	2.03	2.6	3.1	4.4(30)	3.4(52)
OSP	80	1.68	1.70	1.58/0.01	0.85	24.1	25.0	8.0(49)/ <120	30.2(50)
JSPTW	22	4.28	3.87	97.88	4.14	24.4	24.3	-(0)	-(0)

Les résultats expérimentaux montrent la faisabilité de l'approche. Le sur-coût d'AEON par rapport à des algorithmes dédiés est faible et le temps d'analyse est très acceptable et croit de manière quadratique avec la taille du problème.

Actuellement, nous travaillons à l'ajout d'une grande variété d'algorithmes pour de nombreuses classes de problèmes d'ordonnancement. A plus long terme, notre recherche va se concentrer dans deux directions. D'abord, la linéarisation automatique de modèles pour pouvoir utiliser des solveurs MIP ou pour obtenir des bornes inférieures par relaxation linéaire. Ensuite, l'intégration de recherches par défaut robustes pour des classes de problèmes générales (par exemple, l'ordonnancement disjonctif) où on trouve des contraintes hétéroclites.

Remerciements

Merci aux relecteurs pour leurs commentaires constructifs. Ce travail est partiellement soutenu par la région Wallonne, projet Transmaze (516207) et par le programme Pôles d'Attraction Interuniversitaire (politique scientifique fédérale belge).

Références

[1] S.B. Akers and J. Friedman. A non-numerical approach to production scheduling problems. *Operations Research*, 3 :429–442, 1955.

[2] M. Dell'Amico and M. Trubian. Applying tabu search to the job-shop scheduling problem. *Annals of Operations Research*, 41 :231–252, 1993.

[3] R. Kolisch and A. Sprecher. Psplib — a project scheduling problem library. *European Journal of Operational Research*, 96 :205–216, 1997.

[4] Philippe Laborie and Daniel Godard. Self-adapting large neighborhood search : Application to single-mode scheduling problems. *In Proceedings MISTA-07, Paris*, 2007.

[5] Ching-Fang Liaw. A tabu search algorithm for the open shop scheduling problem. *Computers and Operations Research*, 26 :109–126, 1999.

[6] Jean-Noël Monette, Yves Deville, and Pascal Van Hentenryck. Aeon : Synthesizing scheduling algorithms from high-level models. *Operations Research and Cyber-Infrastructure*, pages 43–59, 2009.

[7] Wim Nuijten, T. Bousonville, Filippo Focacci, Daniel Godard, and Claude Le Pape. Towards an industrial manufacturing scheduling problem and test bed. *PMS*, 2004.

[8] Philippe Refalo. Impact-based search strategies for constraint programming. In *CP 2004, Toronto (Canada)*, pages 557–571, 2004.

[9] Pascal Van Hentenryck and Laurent Michel. Scheduling abstractions for local search. In *CP-AI-OR'04, Nice*, pages 319–334, 2004.

[10] Pascal Van Hentenryck and Laurent Michel. *Constraint-Based Local Search*. The MIT Press, 2005.

[11] Pascal Van Hentenryck and Laurent Michel. Synthesis of constraint-based local search algorithms from high-level models. *AAAI'07, Vancouver, British Columbia*, 2007.

[12] Pascal Van Hentenryck, Laurent Michel, Philippe Laborie, Wim Nuijten, and Jerome Rogerie. Combinatorial optimization in OPL studio. In *Portuguese Conference on Artificial Intelligence*, pages 1–15, 1999.

[13] Chao Yong Zhang, PeiGen Li, YunQing Rao, and ZaiLin Guan. A tabu search algorithm with a new neighborhood structure for the job shop scheduling problem. *Computers & Operations Research*, 34 :3229–3242, 2007.

[14] Chao Yong Zhang, PeiGen Li, YunQing Rao, and ZaiLin Guan. A very fast ts/sa algorithm for the job shop scheduling problem. *Computers & Operations Research*, 35 :282–294, 2008.

Reformulation de problèmes de satisfaction de contraintes basée sur des métamodèles

Raphaël Chenouard[1], **Laurent Granvilliers**[1] **and Ricardo Soto**[1,2]

[1]LINA, CNRS, Université de Nantes, France

[2]Escuela de Ingeniería Informática, Pontificia Universidad Católica de Valparaíso, Chile

{raphael.chenouard,laurent.granvilliers,ricardo.soto}@univ-nantes.fr

Abstract

Un des challenges importants en programmation par contraintes est la reformulation de modèles déclaratifs en programmes exécutables permettant de calculer leurs solutions. Cette phase peut nécessiter de faire des traductions entre les langages de la programmation par contraintes, de changer la représentation des contraintes ou d'optimiser les modèles et de paramétrer les stratégies de recherche. Dans cet article, nous présentons un métamodèle pivot qui prend en compte les structures communément rencontrées dans les modèles en programmation par contraintes. Ce métamodèle comprend ainsi plusieurs types de contraintes, des structures conditionnelles ou des boucles, mais aussi les concepts de classe et de prédicat. Ce métamodèle est suffisamment général pour s'adapter aux structures de la plupart des langages, comme les langages orientés-objet ou les langages basés sur la logique, tout en en restant indépendant. Les opérations de reformulation traitent des instances du métamodèle pivot indépendamment des langages contraintes. Ainsi, les opérations définies sont génériques et s'appliquent quelques soient les langages choisis. L'espace des langages est relié à l'espace des métamodèles à l'aide d'analyseurs lexicaux et grammaticaux. Les outils de l'ingénierie logicielle peuvent alors être utiliser pour implémenter un tel cadre de transformation pour les modèles contraintes.

1 Introduction

En programmation par contraintes (PPC), l'utilisateur décrit les caractéristiques d'un problème (CSP) comme des contraintes s'appliquant à des variables. Il utilise, ensuite, un solveur pour calculer les solutions du problème qu'il a décrit. La correspondance automatique entre un modèle et un programme exécutable par un solveur est un des points clés abordés dans ce papier. L'objectif est de développer des outils logiciels intermédiaires qui sont capables de reformuler les modèles en tenant comptes des caractéristiques du solveur.

La modélisation de problèmes réels nécessite l'utilisation de langages de modélisation de haut-niveau avec des structures telles que la définition de contraintes, d'instructions de programmation et des possibilités de modularité. Récemment, plusieurs langages de modélisation sont apparus pour répondre aux besoins variés d'utilisateurs et de catégories de problèmes. D'un côté, il y a de nombreux langages de modélisation pour les problèmes combinatoires tel qu'OPL [19], Essence [5] et MiniZinc [14] ou pour les problèmes numériques comme Numerica [18] et Realpaver [9]. D'un autre côté, des librairies de résolution par contraintes ont été intégrés dans des langages de programmation informatique, comme pour ILOG Solver [15], Gecode [16] et ECL^{i}PS^e [1]. Par la suite, nous allons principalement considérer les langages de modélisation pour définir les modèles CSP. Cependant, les langages de programmation peuvent être choisis comme cible du processus de traduction. Notre objectif est de fournir un outil de traduction *many-to-many* qui peut prendre en compte des langages variés.

La plupart des langages partagent les mêmes structures, comme par exemple la définition des contraintes. D'autres structures sont plus spécifiques, comme les classes dans les langages orientés-objet ou les prédicats dans les langages logiques. Nous avons intégré l'ensemble de ces concepts dans un métamodèle, c'est-à-dire un modèle des modèles CSP. Ce métamodèle sert de pivot dans notre approche et décrit les relations existantes entre les concepts rencontrés en PPC. Il décrit, de manière abstraite, les règles de la modélisation en PPC, ce qui représente en soi une contribution importante par rapport aux travaux précédents [2] qui étaient restreints à des correspondances *one-to-many*

en partant d'un seul langage : s-COMMA.

Le processus de reformulation que nous proposons se décompose en trois étapes. Durant la première, le modèle définit par l'utilisateur est analysé grammaticalement et un modèle conforme à un métamodèle est généré. Durant la dernière étape, le programme résultant est obtenu à l'aide du processus inverse en reformulant l'information présente dans un modèle en respectant la grammaire du langage visé. Ces deux étapes établissent un pont entre l'espace des grammaires et l'espace des modèles. L'étape intermédiaire implémente des opérations de reformulation sur des modèles conformes au métamodèle pivot pour, par exemple, reformuler un modèle basé sur des entiers en un modèle booléen. L'un des points clés de notre approche est de manipuler les concepts CSP avec leur sémantique et non pas des éléments syntaxiques. Ainsi, les opérations de reformulation sont définies sans être restreintes à un langage donné.

Le travail sur la plateforme Cadmium [4] est le plus proche de ce que nous présentons. Cette plateforme utilise un langage de programmation à base de règles combinant les *Constraint Handling Rules* [7] et les opérations de réécriture de termes pour transformer des modèles contraintes formuler en Zinc ou MiniZinc. L'algorithme de réécriture fait la correspondance entre des règles et des termes pour générer de nouveaux termes, jusqu'à obtenir un ensemble de termes sur lequel plus aucune règle n'agit. Cette approche fournit une sémantique claire sur la procédure de transformation, tout en adressant des problèmes de confluence et de terminaison. La définition de métamodèles nous permet d'utiliser les outils de l'ingénierie des modèles comme ATL [12], qui est un langage général de transformation à base de règles mixant des règles déclaratives s'appliquant à des éléments typés et des structures de programmation impératives. Kermeta [13] est une autre plateforme de transformation, qui se base plus sur les concepts de la programmation orientée-objet. Un des bénéfices de l'approche dirigée par les modèles est de directement manipuler des éléments typés conformément à des concepts, qui sont reliés et hiérachisés dans des métamodèles.

La section suivante de l'article présente notre plateforme générale de transformation de modèles appliquée à la PPC. Dans la section 3, un exemple illustrant l'intérêt de l'approche est présenté en se basant sur des langages connus en PPC. La section 4 intègre une présentation du métamodèle pivot ainsi qu'une description des opérations de reformulation. Des expérimentations sur des problèmes couramment rencontrés en PPC sont présentées en section 5. Enfin, la section 6 fait le bilan de la contribution proposée dans cet article et détaille de futurs pistes de recherche.

2 Une plateforme dirigée par les modèles

Un modèle CSP est une représentation d'un problème, écrite dans un langage et ayant une structure. Notre objectif est de transformer des modèles indépendants des solveurs vers des modèles dépendants des solveurs. Cela implique de :

- changer la formulation des constraintes pour améliorer la résolution.
- traduire des modèles écrit dans des langages à haut-niveau niveau d'abstraction vers des langages bas-niveau acceptés par les solveurs ou vers des langages de programmation.
- modifier la structure des modèles par rapport aux caractéristiques des solveurs. Par exemple, il peut être nécessaire de passer d'un modèle orienté-objet à un modèle logique basé sur des prédicats.

La gestion des représentations des contraintes nécessite de spécifier des règles de transformation permettant d'obtenir une formulation équivalente ou des relaxations. La traduction des langages requiert la définition de correspondances entre les syntaxes concrètes et les concepts des métamodèles. La manipulation des structures concerne, en particulier, les concepts de modélisation abstraits comme les objects ou les prédicats. Une motivation importante de ce travaille est de séparer ces différentes tâches. En effet, l'équivalence de formulations des contraintes est indépendante d'un langage spécifique. Le processus de traduction fait alors appel à deux espaces techniques : (1) celui des grammaires (Grammar TS) qui se rapporte aux problèmes liées aux langages et leur syntaxe, (2) celui de l'ingénierie des modèles (MDE TS) qui permet la définition des concepts de modélisation et des règles de transformation à appliquer sur les modèles (cf. figure 1).

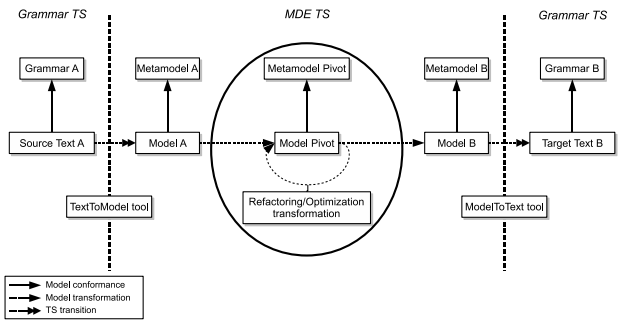


FIGURE 1 – Le processus de transformation des modèles contraintes.

Dans l'espace des grammaires, des termes sont considérés et sont organisés en fonction des règles définies dans une grammaire. Dans l'espace des modèles, les éléments considérés sont typés par rapport à un concept défini dans un métamodèle et correspondent à une instance de ce concept. Les éléments sont alors reliés entre eux à l'aide de simples références ou d'un lien de composition. Par exemple, une contrainte $x + y = z$ est issue du concept de contrainte algébrique composé d'une expression algébrique. Ainsi, il est possible de définir des éléments complexes comme des systèmes de contraintes composés d'ensemble de contraintes.

Le passage des langages aux modèles peut être implémenté à l'aide de techniques d'analyse lexical et grammaticale. Un modèle A est créé à partir d'un fichier source définit par l'utilisateur. Ce modèle doit être conforme au métamodèle du langage choisi par l'utilisateur, comme requis par l'espace des modèles. En conséquence, les métamodèles des langages (les langages de modélisation, les langages de PPC et les langages des solveurs) doivent être définis pour pouvoir établir le type des modèles. La sortie B est générée à partir d'un modèle B. Dans notre approche, ce modèle se conforme à un métamodèle de langage de solveur.

Les transformations de modèles sont définies dans l'espace des modèles. L'objectif est de transformer un modèle A correspondant au modèle écrit par l'utilisateur vers un modèle B associé à un solveur. Comme mentionné précédemment, il est nécessaire de changer la représentation des modèles et leur structure. Ce processus peut être mis en oeuvre à l'aide d'opérations transformant les concepts de A vers ceux de B. Afin d'exploiter les similarités qui peuvent exister entre les langages de modélisation PPC, nous proposons d'introduire un métamodèle intermédiaire servant de pivot. La chaine de transformation est alors composée de trois étapes : (1) le passage d'un modèle A vers le pivot, (2) l'application d'opérations de reformulation sur le modèle pivot et (3) le passage d'un modèle pivot à un modèle B. Ainsi, les étapes de reformulation nécessaire pour un langage donné peuvent être utilisées pour reformuler un autre langage.

Par la suite (section 4), nous allons présenter le métamodèle pivot et les opérations de transformation de modèles. Cependant, nous présenterons d'abord un exemple illustratif et nous discuterons des conditions préalables à la manipulation de modèles contraintes.

2.1 Un exemple illustratif

Nous avons choisi l'exemple des golfeurs sociables pour illustrer le processus de transformation. Le modèle initial est écrit à partir du langage de modélisation orienté-objet s-COMMA. Le modèle cible est un programme écrit dans le langage de programmation logique ECL^{i}PS^e. Dans ce problème, un ensemble de $n = g \times s$ joueurs veulent jouer ensemble au golf chaque semaine. Ils sont répartis dans g groupes de s golfeurs. L'objectif est de trouver un planning permettant aux joueurs de jouer pendant w semaines sans jamais rencontrer deux fois le même joueur. Les figures 2 et 3 présentent le problème écrit en s-COMMA et en ECL^{i}PS^e.

```
//Data file
1. enum Name := {a,b,c,d,e,f,g,h,i};
2. int s := 3; //size of groups
3. int w := 4; //number of weeks
4. int g := 3; //groups per week

//Model file
1. main class SocialGolfers {
2.  Week weekSched[w];
3.  constraint differentGroups {
4.   forall(w1 in 1..w)
5.    forall(w2 in w1+1..w)
6.     forall(g1 in 1..g)
7.      forall(g2 in 1..g) {
8.        card(weekSched[w1].groupSched[g1].players
          intersect
          weekSched[w2].groupSched[g2].players) <= 1;
9.      }
10.   }
11. }
12. class Group {
13.  Name set players;
14.  constraint groupSize {
15.    card(players) = s;
16.  }
17. }
18.
19. class Week {
20.  Group groupSched[g];
21.   constraint playOncePerWeek {
22.    forall(g1 in 1..g)
23.     forall(g2 in g1+1..g) {
24.      card(groupSched[g1].players
          intersect groupSched[g2].players) = 0;
25.     }
26.   }
27. }
```

FIGURE 2 – Un modèle s-COMMA pour le problème des golfeurs sociables.

Le modèle s-COMMA est divisé en deux parties : un fichier de données et un fichier définissant le modèle. Le fichier de données comporte la définition de l'énumération représentant les noms des golfeurs et trois constantes fixant les dimensions du problème (la taille des groupes, le nombre de semaines et le nombre de groupes par semaine). Le fichier du modèle est divisé en trois classes : une pour définir les groupes, une pour les semaines et la classe principale pour définir le problème global des golfeurs sociables. La classe `Group` contient l'attribut `players` qui correspond à l'ensemble des golfeurs qui doivent jouer ensemble. Chaque golfeur est identifié par un nom qui est défini dans l'énumération du fichier de données. Dans cette classe, le bloc appelé `groupSize` (lignes 14 à 16) est un bloc de contraintes, qui permet de regrouper des contraintes s'appliquant aux attributs de la classe. Ces contraintes peuvent être formulées à l'aide de boucles ou de struc-

tures conditionnelles. Ainsi, le bloc mentionné restreint la taille du groupe de golfeurs. La classe `Week` comporte un tableau d'objets de type `Group` et un bloc de contraintes `playOncePerWeek` qui empêche les joueurs de faire parti de plusieurs groupes pour une même semaine. Enfin, la classe `SocialGolfers` contient un tableau d'objets de type `Week` et un bloc de contraintes `differentGroups` qui définit le fait que chaque golfeur ne peut rencontrer qu'une fois le même golfeur tout au long des semaines considérées.

```
1. socialGolfers(L):-
2.  S $= 3,
3.  W $= 4,
4.  G $= 3,
5.
6.  intsets(WEEKSCHED_GROUPSCHED_PLAYERS,12,1,9),
7.  L = WEEKSCHED_GROUPSCHED_PLAYERS,
8.
9.  (for(W1,1,W),param(L,W,G) do
10.  (for(W2,W1+1,W),param(L,G,W1) do
11.   (for(G1,1,G),param(L,G,W1,W2) do
12.    (for(G2,1,G),param(L,G,W1,W2,G1) do
13.      V1 is G*(W1-1)+G1,nth(V2,V1,L),
14.      V3 is G*(W2-1)+G2,nth(V4,V3,L),
15.      #(V2 /\ V4, V5),V5 $=< 1
16.    )
17.   )
18.  )
19. ),
20.
21. (for(I1,1,W),param(L,S,W,G) do
22.  (for(I2,1,G),param(L,S,W,G,I1) do
23.    V6 is G*(I1-1)+I2,nth(V7,V6,L),
24.    #(V7, V8), V8 $= S
25.  )
26. ),
27.
28. (for(I1,1,W),param(L,G) do
29.  (for(G1,1,G),param(L,G,I1) do
30.   (for(G2,G1+1,G),param(L,G,I1,G1) do
31.     V9 is G*(I1-1)+G1,nth(V10,V9,L),
32.     V11 is G*(I1-1)+G2,nth(V12,V11,L),
33.     #(V10 /\ V12, 0)
34.   )
35.  )
36. ),
37.
38. label_sets(L).
```

FIGURE 3 – Le problème des golfeurs sociables exprimés dans le langage ECL^{i}PS^e.

Le modèle ECL^{i}PS^e de la figure 3 est généré après avoir appliqué le processus de transformation présenté dans la section précédente. Un seul prédicat est déclaré et contient l'ensemble du problème. Les dimensions du problème sont d'abord déclarées (lignes 2 à 4), suivies de la liste des variables (des ensembles d'entiers) `L` (lines 6 à 7) correspondant aux joueurs de golf. Enfin, trois blocs de boucles sont définis suite à la transformation des contraintes des trois classes (lignes 9 à 36). Certaines parties de ces deux modèles sont clairement similaires, ce qui se répercute au niveau des métamodèles de ces langages qui partagent un grand nombre de concepts, comme la notion de boucles ou de contraintes. Cependant, les syntaxes pour exprimer ces modèles sont très différentes. Ainsi, la déclaration de boucle `for` en ECL^{i}PS^e nécessite l'utilisation du mot-clé `param` pour définir les variables à importer dans le contexte interne à la boucle.

Le traitement des objets est plus subtile, puisque ce concept n'existe pas en ECL^{i}PS^e. Plusieurs stratégies de transformation sont possibles, comme par exemple, la définition d'un prédicat par classe d'objets [17]. Nous avons choisi une autre approche en enlevant la structure objet des problèmes. L'applatissement d'un problème nécessite d'explorer les hiérarchies d'héritage et les liens de compositions. Plusieurs problèmes peuvent être rencontrés et sont détaillés par la suite. Les attributs peuvent être modifiés de manière conséquente. Par exemple, le tableau `weekSched`, contenant des objets de type `Week` défini à la ligne 2 du fichier modèle de la figure 2, est transformé en une liste plate d'entiers : `WEEKSCHED_GROUPSCHED_PLAYERS` (ligne 6 de la figure 3). Il a aussi été nécessaire d'introduire de nouvelles boucles pour aplatir les tableaux d'objets et ainsi considérer les contraintes de tous les objets déclarés. Le dernier bloc de boucles dans le modèle ECL^{i}PS^e (lignes 27 à 35) a ainsi été généré à partir du bloc de contraintes `playOncePerWeek` du modèle s-COMMA. Il y a une boucle supplémentaire (ligne 27), puisque des instances de `Week` sont contenues dans le tableau `weekSched`. Un autre point à mentionner est la différence d'accès aux éléments des listes entre s-COMMA et ECL^{i}PS^e. Des variables locales ont été introduites et le prédicat Prolog `nth` (équivalent de la contrainte globale `element`) est utilisée dans le modèle ECL^{i}PS^e.

3 Modèle pivot

Nous avons défini un métamodèle pour définir des servant de pivot dans le processus de transformation. Ce métamodèle capture la majeur partie des concepts rencontrés dans les langages contraintes. Les modèles conforment à ce métamodèles sont manipulés à l'aide de transformation de modèles raffinant leur structure et leur formulation. Chaque transformation implémente une unique opération de raffinage ou d'optimisation des modèles.

3.1 Le métamodèle du pivot

La figure 4 présente un extrait de la structure du métamodèle pivot en s'appuyant sur un formalisme de diagramme de classe UML simplifié. Le concept racine est le concept de `Model` qui contient l'ensemble des entités. Trois concepts spécialisent la classe abstraite `ModelElement` :

- `Classifier` représente tous les types qui peuvent être utilisés pour définir des variables ou des constantes :
  - `DataType` correspond aux types de données courants qui sont utilisés en PPC : booléen, entier et réel.
  - `Enumeration` sert à définir des types symboliques, c'est-à-dire basés sur des ensembles de

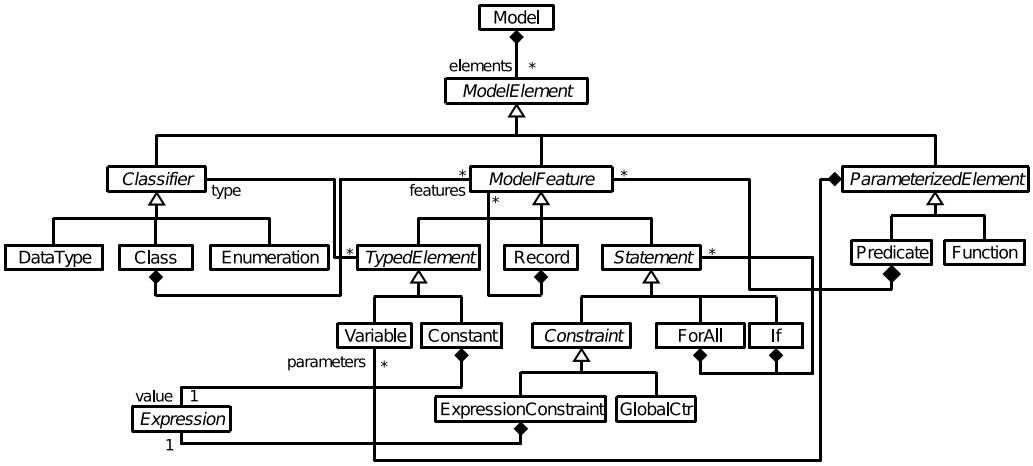


FIGURE 4 – Structures des problèmes et représentation des variables dans le métamodèle pivot.

valeurs symboliques (non détaillées ici, cf. `enum Name :={a,b,...}`, ligne 1 du fichier de données de la figure 2).

- `Class` est similaire au concept de classe défini dans les langages de programmation objet, mais dans le contexte de la PPC [17]. Ainsi une classe comporte des déclarations d'attributs (variables ou constantes), mais aussi des contraintes ou d'autres instructions encapsulant des contraintes (boucle ou conditionnelle). Ces éléments sont regroupés sous le concept abstrait de `ModelFeature`.

– `ModelFeature` correspond à l'ensemble des instances quil est possible de déclarer dans un modèle ou une classe :
  - `Record` se rapporte à des instances non-typées contenant une collection d'éléments à la manière des tuples. Pour étendre la portée des `Record`, nous les avons définis comme une composition d'instances de `ModelFeature`.
  - `TypedElement` est abstrait et regroupe l'ensemble des éléments auxquels nous associons un type (cf. l'association avec le concept `Classifier` sur la figure 4). Le concept de tableau n'est pas dissocié du concept de variable. Un tableau est uniquement défini par des tailles qui correspondant à chacune de ses dimensions. Pour plus de flexibilité, ces tailles sont définies comme des instances d'`Expression`.
    - `Variable` comporte un domaine optionnel (non détaillé ici) qui retreint les valeurs associées à son type. Trois sous-types de `Domain` sont pris en compte : les intervalles, les ensembles et les domaines définis comme des expressions.
    - `Constant` est composé d'une valeur définie comme une expression constante.
  - `Statement` est utilisé pour représenter toutes les autres caractéristiques qui peuvent apparaître dans un modèle ou une classe :
    - `Constraint` est un concept abstrait qui est spécialisé de deux manières : `ExpressionConstraint` correspond aux contraintes définies à l'aide de termes et de relations. `GlobalCtr` correspond aux contraintes globales identifiées par un nom et une liste de paramètres.
    - `ForAll` définit le concept de boucle sur des instances de `Statement`. Une variable locale est alors nécessaire pour définir les itérations de la boucle.
    - `If` permet de définir des conditions pour la prise en compte d'instances de `Statement`. Une expression définit le test booléen. Deux blocs d'éléments sont possibles, le premier étant obligatoire contrairement au deuxième.
– `ParameterizedElement` correspond aux concepts ayant une liste de paramètres et ne correpondant pas à une définition de type ni une `ModelFeature` :
  - `Predicate` permet de déclarer des prédicats logiques comme dans les modèles ECL^{i}PS^e. Les prédicats ont des paramètres et sont composés d'une séquence de `ModelFeature` (par exemple, des variables ou des contraintes).
  - `Function` permet à un utilisateur de définir des fonctions utilisées ensuite dans le modèle. Contrairement aux prédicats, une fonction ne contient qu'une seule instruction.

La notion d'expression est omniprésente en PPC. Les concepts correspondant dans le métamodèle pivot sont détaillées dans la figure 5. Ils représentent les entités apparaissant dans les expressions du premier ordre basées sur des variables, des termes, des relations et des opérateurs.

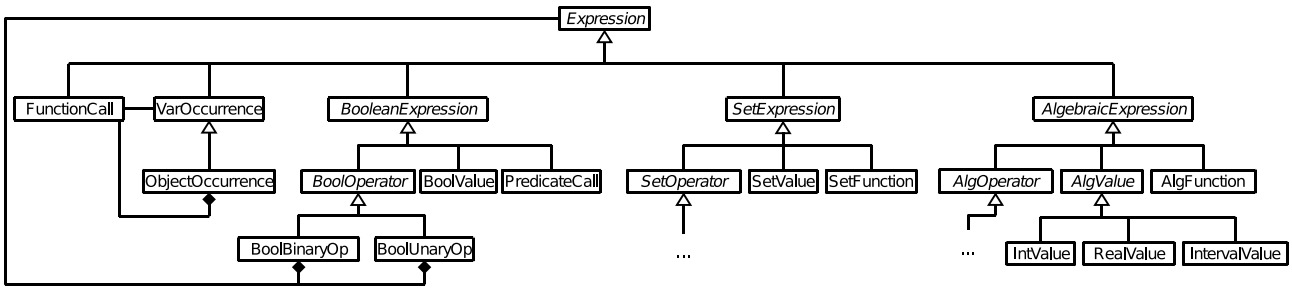


FIGURE 5 – Représentation des expressions utilisées pour définir des contraintes dans le métamodèle pivot.

Le concept `Expression` est abstrait et est spécialisé en plusieurs sortes d'expressions :

- `FunctionCall` fait référence aux fonctions définies. Il comporte une liste de paramètres définis à leur tour comme des expressions.
- `VarOccurrence` sert à représenter l'occurence des variables précédemment déclarées. Tous les éléments ayant un nom sont concernés par ce concept. Lorsque l'élément référencé est un tableau, alors la référence doit avoir un ou plusieurs attributs supplémentaires correspondant aux indices d'accès au tableau. Le concept d'`ObjectOccurrence` spécialise `VarOccurrence` en définissant l'occurence d'un attribut d'objet (par exemple `groupSched[g1].players`, ligne 8 dans la figure 2). Les occurences de variable ne sont pas classifiées par rapport à leur type pour éviter trop de redondances de concept.
- `BooleanExpression` représente globalement les expressions booléennes :
  - `BoolValue` représente les valeurs booléennes `true` et `false`.
  - `PredicateCall` est similaire à `FunctionCall` mais fait référence à des prédicats.
  - `BoolOperator` est abstrait et représente les opérateurs communéments utilisés dans les expressions booléennes : $\neg$, $\Leftrightarrow$, $\rightarrow$, and, or, $=$, $\neq$, $\leq$, $\geq$, $<$, $>$.
- `SetExpression` permet de représenter des expressions ensemblistes en regroupant les valeurs ensemblistes (par exemple $\{1, 2, 3\}$), les fonctions associées (par exemple la fonction de cardinalité) et les opérateurs usuellement utilisés (intersection, union et différence).
- `AlgebraicExpression` représente les expressions numériques sur les entiers ou les réels en utilisant aussi les fonctions et opérateurs classiques ($+$, $-$, $*$, $/$ and $\hat{}$, etc.).

Nous avons définit le métamodèle pivot pour satisfaire la plupart des besoins de modélisation rencontrés en PPC, mais aussi pour correspondre avec les métamodèles des langages PPC. De cette manière des simplifications ont été faites sur ce métamodèle, comme pour le cas des occurences de variables qui ne sont pas définies pour chaque sorte d'expression.

3.2 Reformulation du pivot

Les transformations de modèles sont implémentées commes des opérations de reformulation sur les modèles pivots. Pour plus de clareté, nous ne présentons que quelques opérations principales en utilisant un style de pseudo-code impératif, même si en pratique nous utilisons un langage dédié aux transformations de modèles. L'intérêt principal qui est apporté par la hiérarchie de concepts est le mécanisme de navigation dans les modèles. Par exemple, il est aisé de parcourir l'ensemble des variables d'une contrainte, puisque cette information est décrite dans le concept abstrait de contrainte.

3.2.1 Aplatissement des objets

Cette étape de reformulation remplace les intances correspondant à des objets (les variables dont le type est une classe) par tous les éléments définis dans la classe (variables, constantes, contraintes et autres déclarations). Pour éviter tout conflit de nom, les éléments nommés sont préfixés avec le nom de l'objet.

L'algorithme 1 présente cette étape de transformation écrite en pseudo-code. Ainsi, la fonction `ObjectRemoval` traite un modèle source en itérant sur tous ses éléments (ligne 2). Si des objets sont détectés (ligne 3), alors la fonction `flatten` est appelée et son résultat est ajouté au modèle de sortie (ligne 4). Les éléments qui ne sont pas des objets sont simplement recopiés dans le modèle de sortie (ligne 5 et 6), alors que les classes ne sont pas traitées et donc éliminées. Dans la fonction `flatten` chaque élément de l'ensemble de `ModelFeature` passé en paramètre est dupliqué et ajouté à l'ensemble résultat. Dans le cas des variables (et des constantes), leur nom est redéfini en le préfixant du nom de l'objet passé en premier paramètre (ligne 6). La figure 6 montre le résultat de cette transformation sur l'exemple des golfeurs sociables.

La reformulation des tableaux d'objets et des expressions n'est pas présenté ici pour garder un algorithme

simple. Dans le cas des tableaux d'objets (cf. fin de la section 3), nous devons transférer la définition des tailles du tableau aux attributs des objets et nous ajoutons une déclaration de boucle pour itérer sur l'ensemble des éléments de type `Statement` qui sont encapsuler dans les objets. Au sein des expressions, les occurences de variables doivent simplement être mise à jour pour référencer les nouvelles variables aplaties.

Algorithm 1 Transformation et élimination des variables objets et des classes.

objectRemoval(m : `Model`)
: `Model`

1: **let** res : `Model`
2: **for all** o **in** m.elements **do**
3: **if** is_var(o) **and** is_class(o.type) **then**
4: res.insert(**flatten**(o,o.type.features))
5: **else if not** $is\_class(o)$ **then**
6: res.elements.insert(o)
7: **end if**
8: **end for**
9: **return** res

flatten(o : `Variable`, features : **Set** of `ModelFeature`)
: **Set** of `ModelFeature`

1: **let** res : **Set** of `ModelFeature` = $\emptyset$
2: **for all** f **in** features **do**
3: **if** is_var(f) and **not** is_class(f.type) **then**
4: **let** v : `Variable`
5: v ← duplicate(f)
6: v.name = o.name + '_' + v.name
7: res.insert(v)
8: **else**
9: ...
10: **end if**
11: **end for**
12: **return** res

```
class SocialGolfers { Week weekSched[w];...}
class Week { Group groupSched[g];...}
class Group { Name set players;...}
⇒ Name set weekSched_groupSched_players[g*w];
```

FIGURE 6 – Résultat de l'application de l'aplatissement des objets sur l'exemple des golfeurs sociables présenté en s-COMMA.

3.2.2 Elimination de la contrainte Alldifferent

Les contraintes globales ne sont pas gérées par tous les solveurs, ce qui nous a poussés à définir des opérations de reformulation ou de relaxation pour ces contraintes. Nous présentons ici plusieurs opérations distinctes pour la contrainte *alldifferent*($x_1, ..., x_n$). Nous supposons que le domaine de chaque variable x_i est compris entre 1 et n pour simplifier la définition des deux dernières transformations présentées. Nous présentons ainsi trois possibilités de reformulation pour cette contrainte globale, qui est alors remplacée par :

– Un ensemble d'inégalités (voir Algorithme 2). Pour chaque combinaison de variables (lignes 2 et 3), une contrainte est générée et ajoutée au résultat (ligne 6).

Algorithm 2 Transformation de alldifferent en un ensemble d'inégalités

AllDiffToDisequalities(c : `GlobalConstraint`)
: **Set** of `Constraint`

1: **let** res : **Set** of `Constraint` = $\emptyset$
2: **for all** i **in** 1..c.parameters.size() **do**
3: **for all** j **in** $i+1$..c.parameters.size() **do**
4: let x : `Variable` = c.parameter[i]
5: let y : `Variable` = c.parameter[j]
6: res.insert(**new** `Constraint`(x $\neq$ y))
7: **end for**
8: **end for**
9: **return** res

– Une relaxation (voir Algorithme 3. Seulement une contrainte est générée (ligne 3). Elle calcule la somme des valeurs des variables, qui doit alors être égale à $n(n+1)/2$.

Algorithm 3 Génération d'une relaxation de alldifferent

AllDiffToRelaxation(c : `GlobalConstraint`)
: `Constraint`

1: **let** n : Integer = c.parameters.size()
2: **let** sum : `Expression` = $\sum_{i=1}^{n}$ c.parameters[i]
3: **return new** `Constraint`(sum = n(n+1)/2)

– Une version booléenne (voir Algorithme 4). Dans ce cas, nous définissons une matrice de variables booléennes (lignes 2 à 4), où $b[i, j]$ est vraie lorsque x_i a pour valeur j. La ligne 7 vérifie qu'une seule valeur est définie pour chaque variable. La ligne 10 assure ensuite que deux variables ont une valeur différente.

4 Expérimentations

L'architecture présentée a été implémentée avec trois outils et langages issus de l'ingénierie des modèles : (1) KM3 [10] est un langage permettant de définir des métamodèles, (2) ATL [12] est une langage déclaratif à base de règles pour décrire des transformations de modèles et (3) TCS [11] est un langage déclaratif permettant de lier une grammaire et un métamodèle. Ces outils nous permettent de choisir les opérations de reformulation à appliquer sur

Algorithm 4 Reformulation de alldifferent en modèle booléen

AllDiffToBoolean(c : `GlobalConstraint`)
: **Set** of `ModelFeature`

1: **let** res : **Set** of `Constraint` = $\emptyset$
2: **let** n : Integer = c.parameters.size()
3: **let** m : Integer = card(c.parameters.domain)
4: **let** b[n,m] : Boolean
5: res.insert(b)
6: **for** i **in** $1..n$ **do**
7: res.insert(**new** `Constraint`($\sum_{j=1}^{m}$ b[i,j] = 1))
8: **end for**
9: **for** j **in** $1..m$ **do**
10: res.insert(**new** `Constraint`($\sum_{i=1}^{n}$ b[i,j] = 1))
11: **end for**

les modèles du pivot. Nous pouvons ainsi choisir de garder ou de supprimer la structure suivant le métamodèle cible.

Nous avons mené une série de tests pour analyser les performances de notre approche. Nous avons utilisé cinq problèmes connus en PPC : les golfeurs sociables (Golfers), la conception d'un moteur (Engine), Send+More=Money, les mariages stables (Marriage) et les 10-reines (10-Q). La première série d'expérimentation présente les performances en termes de temps de traduction, alors que la deuxième montre que la génération automatique de modèle n'impacte pas négativement les temps de résolution. Les test ont été effectués sur un ordinateur à 2.66 Ghz avec 2 Go de mémoire ram sous Ubuntu.

Problems	sC Lines	s-to-P (s)	Object (s)	Enum (s)	P-to-E (s)	Total (s)	Ecl Lines
Golfers	31	0.276	0.340	0.080	0.075	0.771	37
Engine	112	0.292	0.641	0.146	0.087	1.166	78
Send	16	0.289	0.273	-	0.089	0.651	21
Marriage	46	0.330	0.469	0.085	0.067	0.951	26
10-Q	14	0.279	0.252	-	0.033	0.564	12

TABLE 1 – Temps obtenus pour une chaine de transformation complète sur plusieurs exemples classiques.

Dans ce premier test, nous présentons une chaine de transformation de s-COMMA (sC) vers ECL^{i}PS^e (Ecl). La table 1 présente les résultats obtenus, où la première colonne correspond aux noms des problèmes et la deuxième le nombre de lignes des fichiers sources. Les colonnes suivantes présentent les temps mesurés pour appliquer les étapes atomiques de la chaine de transformation en secondes : de s-COMMA vers le Pivot (s-to-P), l'aplatissement d'objets (Object), l'élimination des énumérations (Enum) et la transformation du pivot vers ECL^{i}PS^e (P-to-E). La colonne suivante montre le temps total de la chaine de transformation et la dernière colonne représente le nombre de lignes des fichiers ECL^{i}PS^e générés.

Les résultats obtenus montrent que les phases de traitement depuis ou vers les langages PPC (s-to-P et P-to-E) sont rapides, même si on peut constater que les problèmes considérés sont assez courts (au maximum 112 lignes). La transformation de s-COMMA vers le pivot est plus lente que la transformation du pivot vers ECL^{i}PS^e. Cela s'explique par les phases de reformulation sur le pivot qui réduisent le nombre d'éléments à traiter pour cette dernière opération. La phase d'aplatissement des objets est la plus coûteuse. C'est l'exemple du moteur qui obtient ainsi les temps de transformation les plus long, car il comporte plusieurs objets encapsulés les uns dans les autres. Sur l'ensemble des phases de transformation, nous pensons que les temps obtenus sont raisonnables, ce qui permet de traiter aisément des problèmes de plus grande taille.

Problems	Native		Generated		Generated (Flat)	
	solve(s)	Lines	solve(s)	Lines	solve(s)	Lines
Golfers	0.21	28	0.21	31	0.22	276
Marriage	0.01	42	0.01	46	0.01	226
20-Q	4.63	11	4.65	12	5.02	1162
28-Q	80.73	11	80.78	12	87.73	2284

TABLE 2 – Temps de résolution et taille des modèles pour des fichiers écrits à la main ou générés automatiquement.

Dans la deuxième série de tests, nous comparons les fichiers ECL^{i}PS^e générés automatiquement par notre approche avec les fichiers ECL^{i}PS^e des mêmes problèmes, mais écrits à la main (cf. table 2). Nous considérons le temps de résolution et le nombre de lignes de chaque fichier. Les résultats des modèles écrits à la main sont d'abord présentés. Ensuite, les fichiers générés en conservant la structure des boucles. Enfin, nous considérons les fichiers générés pour lesquels les boucles ont été déroulées (Flat). Pour les modèles avec des boucles le nombre de lignes et les temps de calculs sont comparables aux modèles écrits à la mains. En ce qui concerne les modèles déroulés, la taille des modèles augmente évidemment très significativement. Cependant, le temps de résolution est peu différent, sauf pour les modèles de plus grande taille (20-Q et 28-Q de 0,4 et 7 secondes). Cet impact négatif peut être attribué à l'algorithme de propagation incrémentale qui est généralement utilisé dans les systèmes CLP. Nous supposons ainsi qu'une propagation est lancée à chaque fois qu'une contrainte est ajoutée au store. Par contre, si une boucle est déclarée, l'ensemble de contraintes induites sont ajoutées d'un bloc et ainsi il n'y a qu'une seule étape de propagation. Lorsque les boucles sont déroulées, alors une phase de propagation est lancée après chaque ajout de contrainte, ce qui explique les pertes de temps entre les deux versions générées des mêmes problèmes. Cet impact négatif sur le temps de résolution montre qu'il peut être préférable de garder la structure des modèles lorsqu'elle est gérée par le solveur visé, plutot que de générer des modèles complètement aplatis.

5 Travaux connexes

La transformation de modèles est un sujet de recherche récent en PPC. Peu d'approches de transformation des modèles PPC ont été proposées. Les structures indépendantes des solveurs disponibles sont les plus proches de notre travail, comme par exemple MiniZinc (and Zinc), Essence and s-COMMA.

MiniZinc est un langage de modélisation haut-niveau qui permet d'obtenir des programmes ECL^{i}PS^e et Gecode. Les transformations sont implémentées dans Cadmium, qui utilise des techniques basées sur la réécriture de termes. Le processus de traduction implique un modèle intermédiaire, où les structures de MiniZinc sont remplacées par celles supportées par tous les solveurs. Cela facilite ensuite la traduction vers les solveurs, tout en optimisant parfois le modèle obtenu.

Essence est une autre langage impliquant des transformations indépendantes d'un solveur. Des modèles pour ECL^{i}PS^e et Minion [8] peuvent être générés. Le processus de transformation est implémenté au sein du système Conjure [6], qui prend en entrée des spécificaiton Essence pour les raffiner dans un modèle intermédiaire Essence'. La transformation de Essence' vers les solveurs se fait alors à l'aide de traducteurs écrits à la main.

s-COMMA est un langage orienté-objet basée sur une plateforme de modélisation indépendante de tout solveur. Des programmes pour ECL^{i}PS^e, Gecode/J, RealPaver et GNU Prolog [3] peuvent être générés. Un langage intermédiaire (Flat s-COMMA) est aussi utilisé pour faciliter les traductions vers les solveurs. Des traducteurs écrits à la main et utilisant des outils de l'ingénierie des modèles sont disponibles.

Notre approche peut être vue comme une évolution naturelle dela plateforme s-COMMA, tout en apportant deux avantages principaux :

- Dans les approches mentionnées précédemment, seul un langage de modélisation peut être utilisé comme source du processus de transformation, ce qui n'est pas le cas dans notre approche. Nous pensons que cela apporte plus de flexibilité et de liberté pour l'utilisateur.
- Dans les processus de transformation de s-COMMA, MiniZinc, et Essence, toutes les phases de reformulation sont toujours appliquées. Il en résulte des modèles exécutables généralement très différents des modèles initiaux. De notre côté, nous cherchons à générer des modèles optimisés tout en maintenant autant que possible la structure initiale des modèles sources. Nous pensons, que transférer les caractéristiques des modèles sources aux modèles cibles, améliore ainsi la lisibilité et la compréhensibilité des modèles générés.

6 Conclusion et travaux futurs

Dans cet article, nous avons présenté un nouveau cadre pour la transformation des modèles en PPC. Ce cadre est basé sur une approche dirigée par les modèles avec un métamodèle pivot qui nous permet d'être indépendant et flexible tout en s'adaptant aux différents langages de la PPC. La chaine de transformation implique trois principales étapes : (1) du langage source vers le pivot, (2) des opérations de reformulation sur le pivot et (3) du pivot vers le langage cible. Contrairement aux autres approches, les chaines de transformation sont modulaires et le travail de reformulation/optimisation est effectué de manière générique sur le pivot. Les transformations sont plus simples et, par conséquent, l'intégration de nouveaux langages et de nouvelles étapes de reformulation nécessite moins d'efforts.

Nous comptons prochainement intégrer de nouveaux langages à notre approche. La définition de nouvelles opérations de reformulation et d'optimisation des modèles est aussi une de nos priorités. Une autre piste de recherche que nous voulons développer concerne la gestion de chaines complexes de transformation. L'ordre d'application des transformations et leur choix peuvent être automatisés, mais cela nécessite d'analyser statiquement les transformations et les modèles à considérer.

Références

[1] Krzysztof R. Apt and Mark Wallace. *Constraint Logic Programming using Eclipse*. Cambridge University Press, New York, NY, USA, 2007.

[2] R. Chenouard, L. Granvilliers, and R. Soto. Model-Driven Constraint Programming. In *ACM SIGPLAN PPDP*, pages 236–246, 2008.

[3] D. Diaz and P. Codognet. The GNU Prolog System and its Implementation. In *SAC 2000*, pages 728–732, 2000.

[4] Gregory J. Duck, Peter J. Stuckey, and Sebastian Brand. ACD Term Rewriting. In *ICLP*, pages 117–131, 2006.

[5] A. M. Frisch, M. Grum, C. Jefferson, B. Martínez Hernández, and I. Miguel. The Design of ESSENCE : A Constraint Language for Specifying Combinatorial Problems. In *IJCAI*, pages 80–87, 2007.

[6] A.M. Frisch, C. Jefferson, B. Martinez-Hernandez, and I. Miguel. The Rules of Constraint Modelling. In *IJCAI*, pages 109–116, 2005.

[7] T. Frühwirth. *Constraint Handling Rules*. Cambridge University Press, June 2009. to appear.

[8] I. P. Gent, C. Jefferson, and I. Miguel. Minion : A Fast Scalable Constraint Solver. In *ECAI*, pages 98–102, 2006.

[9] L. Granvilliers and F. Benhamou. Algorithm 852 : RealPaver : an Interval Solver Using Constraint Satisfaction Techniques. *ACM Trans. Math. Softw.*, 32(1) :138–156, 2006.

[10] F. Jouault and J. Bézivin. KM3 : A DSL for Metamodel Specification. In *FMOODS*, LNCS 4037, pages 171–185, 2006.

[11] F. Jouault, J. Bézivin, and I. Kurtev. TCS : a DSL for the Specification of Textual Concrete Syntaxes in Model Engineering. In *ACM GPCE*, pages 249–254, 2006.

[12] Ivan Kurtev, Klaas van den Berg, and Frédéric Jouault. Rule-based Modularization in Model Transformation Languages Illustrated with ATL. *Science of Computer Programming,*, 68(3) :138–154, 2007.

[13] Pierre-Alain Muller, Franck Fleurey, and Jean-Marc Jézéquel. Weaving Executability into Object-Oriented Meta-Languages. In S. Kent L. Briand, editor, *Proceedings of MODELS/UML*, LNCS 3713, pages 264–278, Montego Bay, Jamaica, October 2005.

[14] N. Nethercote, P. J. Stuckey, R. Becket, S. Brand, G. J. Duck, and G. Tack. MiniZinc : Towards A Standard CP Modelling Language. In *CP*, LNCS 4741, pages 529–543, 2007.

[15] J.F. Puget. A C++ Implementation of CLP. In *SPICIS*, 1994.

[16] C. Schulte and G. Tack. Views and Iterators for Generic Constraint Implementations. In *Recent Advances in Constraints (2005)*, LNCS 3978, pages 118–132, 2006.

[17] R. Soto and L. Granvilliers. The Design of COMMA : An Extensible Framework for Mapping Constrained Objects to Native Solver Models. In *IEEE ICTAI*, pages 243–250, 2007.

[18] P. Van Hentenryck, L. Michel, and Y. Deville. *Numerica : a Modeling Language for Global Optimization*. MIT Press, 1997.

[19] P. Van Hentenryck, L. Michel, L. Perron, and J.-C. Régin. Constraint Programming in OPL. In *PPDP*, LNCS 1702, pages 98–116, 1999.

Vers une Théorie du Test des programmes à contraintes

Nadjib Lazaar*, Arnaud Gotlieb*, Yahia Lebbah**

* INRIA Rennes Bretagne Atlantique, Campus Beaulieu, 35042 Rennes Cedex, France
** Université d'Oran Es-Senia, B.P. 1524 EL-M'Naouar, 31000 Oran, Algerie
Nadjib.Lazaar@irisa.fr, Arnaud.Gotlieb@irisa.fr, ylebbah@gmail.com

Abstract

Tout processus de développement logiciel effectué dans un cadre industriel inclut désormais une phase de test ou de vérification formelle, y compris pour le développement des programmes à contraintes. Notre travail vise à poser les jalons d'une Théorie du test des programmes à contraintes qui puisse servir de socle à cette vérification. Cette nouvelle théorie est également motivée par le développement récent de plusieurs langages de modélisation de haut-niveau tels que OPL, ZINC, COMET, ou ESSENCE, qui ouvre la voie à des recherches orientées vers les aspects génie logiciel autour de la programmation par contraintes. Notre approche repose sur certaines hypothèses quant au développement et raffinement dans un langage de PPC. Il est usuel de démarrer à partir d'un modèle simple et très déclaratif, une traduction fidèle de la spécification du problème, sans accorder d'intérêt à ses performances. Par la suite, ce modèle est raffiné par l'introduction de contraintes redondantes ou reformulées, l'utilisation de structures de données optimisées, et de contraintes globales. Nous pensons que l'essentiel des fautes introduites est compris dans cette deuxième étape. Dans cet article nous bâtissons les prémisses d'une Théorie du test des programmes à contraintes qui établit les règles de conformité entre le modèle initial déclaratif et le programme optimisé dédié à la résolution d'instances de grande taille. Nous illustrons cette Théorie à l'aide du problème des règles de golomb en OPL et détaillons une première validation expérimentale sur le problème d'ordonnancement des véhicules (POV).

1 Introduction

La Programmation par Contrainte a connu un véritable succès en résolution de problèmes combinatoires dans l'industrie et dans divers domaines (optimisation, transport, emploi du temps, ordonnancement, etc.). D'autres domaines d'application se développent désormais, comme la bio-informatique (Will et Backofen [1]), ou encore la biochimie (Fages et al. [4]). On trouve de nombreux langages et plate-formes PPC commercialisés comme COMET[1], OPL[2], SICStus[3] et académiques tels que CHOCO[4], ESSENCE [5], ZINC [9]. Or, pour les programmes développés dans ces langages, le processus de développement logiciel devrait également inclure une phase de vérification et/ou de validation qui inclut par exemple de la preuve de programme, de la vérification de modèles ("model-checking"), de l'analyse statique ou du test. Pour cette dernière approche dans le cas du test des programmes conventionnels, plusieurs théories ont été proposées telles que la théorie du test de Goodenough et Gerhart [6], la théorie mathématique du Test de Gourlay [7], la Théorie du Test de Weyuker et Ostrand [14].

Pourtant, le test des programmes à contraintes ne peut-être envisagé comme celui des programmes traditionnels. Cela pour deux raisons principales : premièrement, le modèle de faute des programmes à contraintes est différent de celui des programmes traditionnels car l'objectif est ici de calculer des solutions d'un système de contraintes ; deuxièmement, le processus de développement et de raffinement n'est pas le même pour les programmes à contraintes et pour les programmes traditionnels. En effet, il est usuel de démarrer à partir d'un modèle simple et très déclaratif du problème que l'on cherche à résoudre, une traduction fidèle de la spécification du problème, sans accorder d'intérêt à ses performances. Puis, des techniques puissantes de raffinement de modèle sont mises en oeuvre. Par exemple

[1] http ://www.dynadec.com/support/downloads/
[2] http ://www.ilog.com/products/oplstudio/
[3] http ://www.sics.se/isl/sicstuswww/site/
[4] http ://choco.sourceforge.net

un codage approprié du problème à l'aide de structures de données optimisées est proposé, ainsi qu'une reformulation des contraintes d'origine. Des contraintes globales sont utilisées afin de maximiser le pouvoir de déduction du modèle, ainsi que des contraintes redondantes ou des contraintes qui visent à casser des symétries du problème (ces contraintes améliorent considérablement l'efficacité de la résolution). Illustrons ce raffinement sur le problème classique des règles de Golomb. Ces règles trouvent leurs champs d'application dans des domaines variés tels les communications radio, rayons X en cristallographie, les codes convolutionnels doublement orthogonaux, tableaux des antennes linéaires, communications PPM (pulse phase modulation).

Une règle de Golomb [11] peut être définie comme un ensemble de m entiers $0 = x_1 < x_2 < ... < x_m$ tel que les $m(m-1)/2$ différences $x_j - x_i,\ 1 \leq i < j \leq m$ sont distinctes. On dit qu'une telle règle est d'ordre m si elle contient m marques, et qu'elle est de longueur x_m. L'objectif est de trouver une règle de longueur minimale ($minimize\ x[m]$). Une modélisation simple et très déclarative de ce problème est donnée dans la partie (A) de la Fig. 1. La partie (B) de la Fig. 1 présente quant à elle un modèle raffiné optimisé[5] où une structure de donnée a été introduite (matrice), des symétries ont été cassées statiquement, des contraintes redondantes ont été introduites et une contrainte globale posée (alldifferent). Arrivé à ce point, comment être sûr que le modèle de la partie (B) de la Fig. 1 est conforme à celui de la partie (A) ? En effet, ces raffinements de modèle, réalisés par le développeur, peuvent introduire des fautes. Par exemple, mal formuler une contrainte peut conduire à un modèle surcontraint (supprimer des solutions) ou à un modèle dans lequel des solutions ont été ajoutées.

Dans cet article, nous proposons les prémisses d'une Théorie de la conformité des programmes à contraintes. Notre Théorie s'appuie sur la définition de relations de conformité entre les modèles, sur la proposition de définitions concernant la génération de test (donnée de test, jeu de test idéal, critères de test, faute et test de non-conformité). Nous évoquons une méthode de génération automatique de données de test pour les programmes à contraintes que nous validons sur un exemple réaliste : celui de l'ordonnancement de véhicules (POV).

Le reste de cet article est organisé comme suit. La section 2 est un passage en revue de quelques langages à contraintes et de leurs outils de mise au point. La section 3 décrit notre théorie du test des programmes à contraintes avec les notions de conformité. En section 4, un schéma de validation expérimentale de la théorie est présenté sur l'exemple de l'ordonnancement de véhicules. Enfin la section 5 conclut l'article avec des perspectives d'extension de cette Théorie.

[5]On se place dans un contexte où la contrainte globale $Golomb(Var)$ n'est pas disponible.

2 Etat de l'art

Au cours des dernières années, beaucoup de travaux ont été menés autour des langages de modélisation de programmes à contraintes. Ces langages, conçus pour un haut niveau de modélisation, permettent une expérimentation facile avec différents solveurs pour un même problème mais ne présentent pas d'innovations majeures en termes de mise au point des programmes à contraintes. Le langage ESSENCE [5] est un langage formel de spécification de problèmes combinatoires qui s'appuie sur la langue naturelle. Le projet G12 développe le langage ZINC[6] et ses variantes (MiniZinc, FlatZinc) dans le but de créer une plateforme logicielle pour résoudre des problèmes d'optimisation combinatoire réels [9]. ZINC définit des prédicats permettant la réutilisation de code, ainsi que des méthodes de traduction vers des modèles de plus bas-niveau. Néanmoins, ces deux plateformes n'offrent pas en standard d'outils pour aider au test ou à la mise au point des programmes. Le langage OPL (*Optimization Programming Language*)[12] est un langage de modélisation utilisant la programmation mathématique et la programmation par contraintes. Il offre à l'utilisateur un outil de mise au point dans OPL Studio qui détecte des erreurs de syntaxe (oubli du ';' après une instruction par exemple), des erreurs dites "sémantiques" (erreur de nom ou de type par exemple) et des erreurs d'exécution (initialisation d'un tableau de longueur N avec $N+1$ données par exemple). COMET [13] est lui un langage orienté objet implanté en C++ avec un certain nombre d'abstractions innovatrices de modélisation et de contrôle pour la recherche locale. Son outil de mise au point est simplement une interface à `gdb`, le "debugger" standard de la suite `gcc`. On peut citer aussi CHOCO [7] qui est une librairie JAVA pour la modélisation des contraintes qui offre, comme COMET, une interface à un debugger du langage sous-jacent (`jdb`). Mais, les fautes les plus difficiles à trouver, celles liées à la correction des modèles à contraintes, ne sont pas visées et détectées par ces outils.

La mise au point des programmes à contraintes a fait l'objet de nombreux travaux de Recherche, en particulier à l'occasion du projet OADymPPaC[8]. Ceux-ci ont abouti à la définition de modèles de trace génériques

[6]http ://www.g12.cs.mu.oz.au/
[7]http ://www.emn.fr/x-info/choco-solver/doku.php
[8]http ://contraintes.inria.fr/OADymPPaC/

```
int m=...;

dvar int x[1..m] in 0..m*m;

minimize x[m];
subject to {
    forall (i in 1..m-1)
        x[i] < x[i+1];
    forall (i in 1..m, j in 1..m,
           k in 1..m, l in 1..m: (i < j, k < l))
       x[j] - x[i] != x[l] - x[k];
}
```

- A -

```
int m=...;

dvar int x[1..m] in 0..m*m;

tuple indexerTuple {
int i;
int j;
}

{indexerTuple} indexes = {<i, j> | i,j in 1..m : i < j};
dvar int d[indexes];

 minimize x[m];
 subject to {

(1)     forall (i in 1..m-1)
            x[i] < x[i+1];

(2)      forall(ind in indexes)
            d[ind] == x[ind.j]-x[ind.i];
(3)     x[1]=0;
(4)     x[m] >= (m * (m - 1)) / 2;
(5)     allDifferent(all(ind in indexes ) d[ind]);
(6)     x[2] <= x[m]-x[m-1];
(7)     forall(ind1 in indexes, ind2 in indexes, ind3 in indexes :
          (ind1.i==ind2.i)&&(ind2.j==ind3.j)&&(ind1.j==ind3.j)&&
           ( ind1.i < ind2.j < ind1.j))
            d[i,j]=d[i,k]+d[k,j];
(8)      forall(ind1,ind2,ind3,ind4 in indexes :
              (ind1.i==ind2.i)&&(ind1.j==ind3.j)&&
              (ind2.j==ind4.j)&&(ind3.i==ind4.i)&&
              (ind1.i<m-1)&&(3<ind1.j<m+1)&&
              (2<ind2.j<m)&&(1<ind3.i<m-1)&&
              (ind1.i < ind3.i < ind2.j < ind1.j))
 d[ind1]==d[ind2]+d[ind3]-d[ind4];
(9)     forall(i in 2..m, j in 2..m, k in 1..m : i  < j)
            x[i]=x[i-1]+k => x[j] != x[j-1]+k;

       }
```

- B -

FIG. 1 – $M_x(k)$ et $P_x(k)$ des règles de Golomb en OPL.

[3, 8] et à la réalisation d'outils d'observation et d'analyse de traces post-mortem, tels que Codeine pour Prolog, Morphine [8] pour Mercury, ILOG Gentra4CP[9], ou encore JPalm/JChoco. SICStus Prolog intègre désormais un outil d'analyse de trace nommé `fdbg` pour les programmes `clpfd` qui est capable de montrer les étapes de propagation de contraintes ainsi que les réductions de domaines effectués. On peut citer également l'outil CLPGUI [4] qui offre une interface graphique et intuitive à la visualisation des traces. Ces outils aident à comprendre les comportements d'un programme à contraintes et aident à leur optimisation, mais ne sont pas dédiés à la détection de fautes. En effet, celle-ci exige la donnée d'une référence (une spécification ou un oracle) afin de déterminer la divergence entre une implantation et cette référence. Tab. 1 récapitule les fonctionnalités des langages que nous venons de citer. Il est à noter que aucun d'eux ne présentent, à notre connaissance, de Théorie et/ou d'outil de test des programmes à contraintes a proprement parler.

[9] http ://www2.ilog.com/preview/Discovery/gentra4cp/

3 Vers une théorie du test des programmes à contraintes

3.1 Notations et définitions

Dans ce qui suit, nous adoptons les notations suivantes : x dénote un vecteur de variable, x_i dénote une valuation de ce vecteur, $(x\backslash x_i)$ représente la substitution des variables x par x_i, la différence de deux ensembles E et F est notée $E\backslash F \triangleq \{x/(x \in E) \wedge (x \notin F)\}$, la différence symétrique de E et F, se note $E\Delta F \triangleq \{x/\,((x \in E) \wedge (x \notin F)) \vee ((x \in F) \wedge (x \notin E))\} = (E \cup F)\backslash(E \cap F)$.

Un programme à contraintes comprend d'une part un modèle $M_x(k)$, qui est une conjonction de contraintes $C_i(x)$, où k représente l'ensemble des paramètres du modèle (pour les règles de Golomb, il s'agit de l'ordre de la règle tandis que le vecteur de marques représentent les variables) ; et d'autre part une procédure de recherche ou d'optimisation notée *Solve*.

TAB. 1 – les différents langages de modélisation des programmes à contraintes. (PL : prog. linéaire, PPC : prog. par contraintes, SAT : satisfiabilité, RL : recherche locale, MAP : mise au point)

//////////	CHOCO	COMET	ESSENCE	OPL	Sicstus	ZINC
Indépendant des solveurs	Oui	Non	Oui	Oui	Non	Oui
Solveur PL	Non	Oui	Oui	Oui	Oui	Oui
Solveur PPC	Oui	Oui	Oui	Oui	Oui	Oui
Solveur SAT	Non	Oui	Non	Non	Non	Oui
Solveur RL	Oui	Oui	Non	Non	Non	Non
Outil de MAP	jdb	gdb	Non	OPL Studio	fdbg	Non
Plateforme	JAVA	C++	HASKELL	C++	Prolog	Mercury
Dernière version	CHOCO-V2	Comet 1.1	Essence 1.1.0	OPL 6.1.1	Sics Prolog 4.0.5	MiniZinc 0.9

Model $M_x(k)$
$\begin{cases} C_1(x) \\ . \\ . \\ . \\ C_n(x) \end{cases}$
Solve($M_x(k)$)

On considère que $k \in \mathcal{K}$ où $\mathcal{K}$ représente l'ensemble des valeurs possible des paramètres pour lequel le modèle $M_x(k)$ possède des solutions. Notons que le vecteur de variables x peut dépendre de l'instance du modèle considérée. Par exemple, si $k = 3$ dans Golomb, l'instance du modèle cherche une règle avec 3 marques (x=(0, 1, 3)) tandis que pour $k = 4$ elle cherche une règle de 4 marques (x=(0, 1, 4, 6)).

On considère une fonction f qui est une fonction de coût à optimiser sur l'Espace de Recherche. Pour fixer le discours, nous considérons uniquement les problèmes de minimisation ($Minimize(f)$), sachant que les problèmes de maximisation s'obtiennent par symétrie. La Fig. 2 donne un exemple de fonction objectif sur $\mathbb{R}$ où le point x_1 représente un minimum global (une solution optimale) avec $f(x_1) = b$. Cette valeur b est pratiquement difficile à atteindre dans les instances réalistes des problèmes, c'est pourquoi nous nous intéressons dans la Théorie, à la notion de *quasi-optimaux*. En effet un développeur est prêt à concéder, à titre de perte maximale autorisée sur la fonction objectif, un coût strictement positif noté l. Dans la Fig. 2, les points x_0, x_3 représentent des solutions quasi-optimales car les valeurs $f(x_0)$ et $f(x_3)$ ne diffèrent de la valeur optimale que d'une quantité inférieure ou égale à l, tandis que le point x_2 n'est pas une solution quasi-optimale car $f(x_2) > b + l$. Cette perte l est relative au problème traité. Soit M un modèle à contraintes, $sol(M)$ dénote l'ensemble des solutions de M, $quasi_f(M, l)$ dénote l'ensemble des solutions quasi-optimales de M par rapport à une fonction objectif f et une perte l, et $best_f(M)$ dénote l'ensemble des minimums globaux.

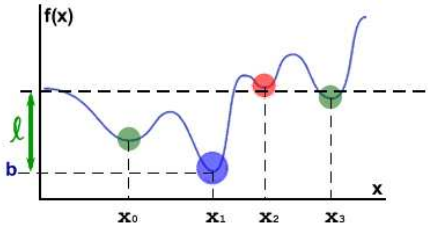


FIG. 2 – Fonction objectif dans le cas univarié.

La vérification de la correction d'un programme implique généralement la présence d'une référence, nommée oracle de test. Cette référence, dans notre approche, est le modèle original et déclaratif du problème. Ainsi, il devient possible d'utiliser ce modèle de plusieurs manières pour générer des données de test, nous y reviendrons dans la suite. A ce point, il est important de noter que le modèle déclaratif devient pour la théorie que nous présentons, une référence incontestée et idéalisée du programme à contraintes sous test. Nous considérons donc ce modèle comme étant correct par rapport aux exigences de l'utilisateur, ce qui est une hypothèse forte mais habituellement faite dans les théories classiques du test des programmes [6]. Nous considérons aussi que ce modèle possède des solutions et caractérise, pour une instanciation des paramètres, toutes les solutions du problème. Si, pour certaines instanciations des paramètres k, le modèle n'a pas de solution, ces valeurs de k sont simplement exclues de l'ensemble de définition $\mathcal{K}$.

Le Modèle–oracle de test est un modèle à contraintes $M_x(k)$ qui caractérise l'ensemble des solutions du problème et qui est conforme aux exigences de l'utilisateur. *Le Programme à Contraintes Sous Test (CPUT)* est un modèle à contraintes $P_x(k)$ qui est l'objet du

test. Le programme $P_x(k)$ vise à résoudre des instances difficiles du problème et il est légitime de le tester avant de l'utiliser pour ces instances car elles peuvent nécessiter un temps de résolution très long ou bien mobiliser des ressources importantes. Il est important de noter qu'étant donné k_0 une instance de k et x_0 un point de l'espace de recherche, vérifier que $M_{(x\backslash x_0)}(k\backslash k_0)$ est vrai est généralement peu coûteux, tandis que trouver x_0 qui satisfait aux contraintes du modèle M est un problème difficile.

Definition 1 (Ensemble $\mathcal{S}$) *On appelle $\mathcal{S}$ l'ensemble des solutions des instances $P_x(k)$:*

$$\mathcal{S} = \bigcup_{k\in\mathcal{K}} sol(P_x(k)).$$

3.2 Relations de conformité

On note *conf* la relation de conformité entre le CPUT $P_x(k)$ et l'oracle de test $M_x(k)$. Nous définissons la relation de conformité *conf* selon le type de problème abordé. En effet, cette définition présente des différences selon que l'on cherche une, toutes ou une meilleure solution d'une instance.

3.2.1 Une seule solution

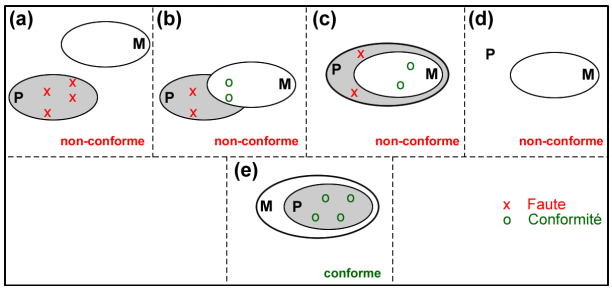


FIG. 3 – $conf_{one}$ Conformité entre $P_x(k)$ et $M_x(k)$ pour une seule solution.

La Fig. 3 présente l'ensemble $sol(M_x(k))$ noté M, et l'ensemble $sol(P_x(k))$ noté P. Les points étiquetés **x** représentent des non-conformités (fautes) tandis que les points étiquetés **o** sont conformes. Les parties (a), (b) et (c) de la Fig. 3 montrent que $P_x(k)$ est non-conforme au Modèle–Oracle $M_x(k)$. En effet, résoudre $P_x(k)$ peut conduire à des solutions qui ne satisfont pas le modèle $M_x(k)$. La partie (d) est aussi une non-conformité car P ne contiens aucune solution. En revanche, la partie (e) montre que $P_x(k)$ est conforme à $M_x(k)$ car elle ne contient pas de points de non-conformité. En observant que P doit être inclus dans M, ce schéma suggère la définition suivante pour la relation de conformité $conf_{one}$:

Definition 2 ***(Relation de conformité : $conf_{one}$)***

$$P\ conf_{one}\ M \Leftrightarrow$$
$$\forall k \in \mathcal{K} :\ sol(P_x(k)) \neq \emptyset \wedge sol(P_x(k)) \subseteq sol(M_x(k))$$

Cette relation de conformité impose que l'ensemble des solutions du programme à contraintes sous test soit non vide. En effet, sans cela, on risque de considérer comme conforme un modèle inconsistant (sans solution). L'ensemble des non-conformités est alors : $sol(P_x(k))\backslash sol(M_x(k))$.

3.2.2 Toutes les solutions

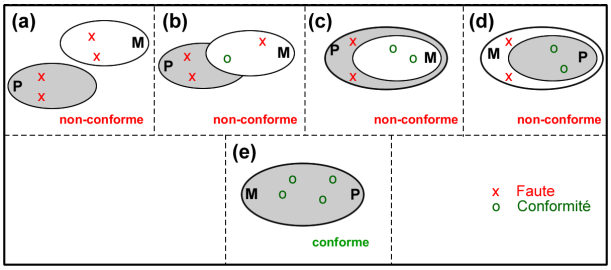


FIG. 4 – $conf_{all}$ Conformité entre $P_x(k)$ et $M_x(k)$ pour toutes les solutions.

On reprend sur la Fig. 4 les ensembles P et M. Dans le cas de recherche de toutes les solutions, les parties (a), (b), (c) et (d) montrent des points de non-conformité. Dans (d), une solution du modèle qui ne serait pas solution du CPUT est une faute. En effet, $P_x(k)$ est conforme à $M_x(k)$ ssi les deux ensembles P et M sont égaux ce qui induit la définition suivante :

Definition 3 ***(Relation de conformité : $conf_{all}$)***

$$P\ conf_{all}\ M \Leftrightarrow \forall k \in \mathcal{K} :\ sol(P_x(k)) = sol(M_x(k))$$

Note : $sol(M_x(k))$ ne peut être vide car k prend ses valeurs dans $\mathcal{K}$.
L'ensemble des non-conformités est alors : $sol(P_x(k))\Delta sol(M_x(k))$.

3.2.3 Une meilleure solution

La Fig. 5 présente la relation de conformité dans le cas où une meilleure solution (quasi-optimale) du CPUT est recherchée. P représente cette fois l'ensemble $quasi_f(P_x(k), l)$, B l'ensemble $best_f(M_x(k))$ qui est l'ensemble des minimums globaux de $M_x(k)$. L'ensemble Q représente $quasi_f(M_x(k), l)$ où les solutions quasi-optimales sont incluses. On choisit quatre solutions quasi-optimales de P. La partie (a) est un cas

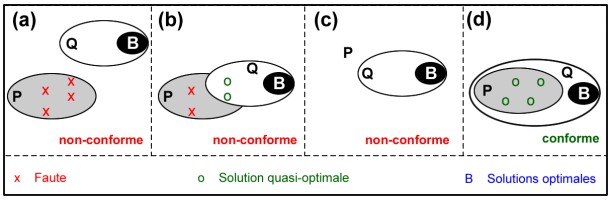


FIG. 5 – $conf_{best}$ Conformité entre $P_x(k)$ et $M_x(k)$ pour une meilleure solution.

de non-conformité car les quasi-optimaux du CPUT ne sont pas inclus dans les quasi-optimaux du modèle-oracle. La partie (b) est un autre cas de non-conformité car au moins deux solutions quasi-optimales ne sont pas dans Q. Le cas (c) présente aussi une non-conformité car P ne contiens aucune solution quasi-optimale ; cela signifie que la minimisation n'a pas pu atteindre une solution avec un coût $\leq l + b$. En revanche, la partie (d) montre un cas de conformité car les quasi-optimaux de P sont dans Q. Ceci suggère une définition fondée sur l'appartenance des solutions quasi-optimales de $P_x(k)$ aux solutions quasi-optimales du modèle-oracle $M_x(k)$:

Definition 4 ***(Relation de conformité :*** $conf_{best}$***)***

$P\ conf_{best}\ M \Leftrightarrow$

$\forall k \in \mathcal{K}$:
$quasi_f(P_x(k), l) \neq \emptyset \wedge$
$quasi_f(P_x(k), l) \subseteq quasi_f(M_x(k), l)$

où l est la perte maximale autorisée sur la fonction objectif f.

L'ensemble des non-conformités est alors : $quasi_f(P_x(k), l) \backslash quasi_f(M_x(k), l)$.

La question naturelle qui se pose est celle de la preuve de conformité : peut-on prouver la conformité du CPUT par rapport à son modèle-oracle ? Si on se place dans le cadre de la résolution des contraintes à domaines finis, prouver la conformité d'une instance est un problème NP_difficile car cela nécessite au moins de trouver toutes les solutions du CPUT. Il nous faut donc réduire cette ambition à la recherche de non-conformité par le test, ce qui justifie l'introduction d'une Théorie du test des programmes à contraintes.

3.3 Test de non-conformité

Nous introduisons ici les définitions permettant d'exprimer la non-conformité.

Definition 5 (Donnée de Test) *Etant donné un modèle-oracle $M_k(x)$, une donnée de test est définie comme une paire (k_0, x_0).*

Definition 6 (Jeu de Test) *Un jeu de test, noté T est un ensemble fini non nul de données de test, tq :*

$$T \subseteq \mathcal{K} \times \mathcal{S}$$

où $\mathcal{S}$ est défini dans Def.1.

Soit $T_d \subseteq \mathcal{K}$, où $T_d = \{k_1, k_2, ...k_l\}$, et $S_i = sol(P_x(k_i))$, on note alors :

$$T \subseteq \mathcal{D} = \{(k_i, s_{ij})\ /k_i \in \mathcal{K}, s_{ij} \in S_i\}$$

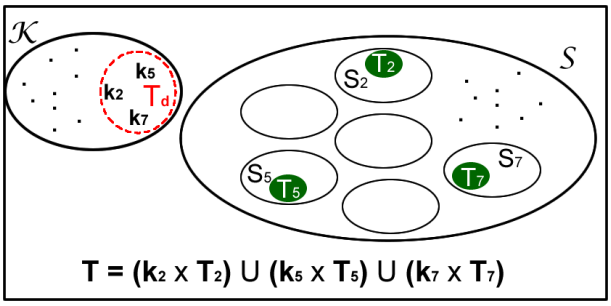


FIG. 6 – Exemple de jeu de test " T ".

La Fig. 6 montre un exemple de jeu de test, où $\mathcal{K} = \{..., k_2, ..., k_5, ..., k_7, ...\}$ donne un ensemble d'instances de $P_x(k)$. L'ensemble $\mathcal{S}$ contient les ensembles de solutions, un pour chaque instance. Si $T_d = \{k_2, k_5, k_7\}$, alors on a trois ensembles de solutions $sol(P_x(k_2))$, $sol(P_x(k_5))$, $sol(P_x(k_7))$, desquels on extrait trois sous-ensembles T_2, T_5, T_7 pour former le jeu de test :

$$T = (k_2 \times T_2) \cup (k_5 \times T_5) \cup (k_7 \times T_7).$$

La qualité d'un jeu de test peut être évaluée par sa force à détecter des non-conformités. Si ces dernières existent, le jeu de test idéal les fera ressortir.

Definition 7 (Jeu de test idéal) *T est un jeu de test idéal ssi :*
$\forall (k_i, x_i) \in T, P_{(k \backslash k_i)}(x \backslash x_i)\ conf_\alpha\ M_{(k \backslash k_i)}(x \backslash x_i) \Rightarrow$
$\forall (k_j, x_j) \in \mathcal{D}, P_{(k \backslash k_j)}(x \backslash x_j)\ conf_\alpha\ M_{(k \backslash k_j)}(x \backslash x_j)$
où α vaut one, all, ou best

L'intérêt de cette définition provient de la remarque suivante : l'ensemble T est un jeu de test pratique, il est donc fini, tandis que l'ensemble $\mathcal{D}$ est en général infini. C'est tout l'intérêt du Test que de pratiquer ce genre de réduction, au prix de la complétude. En pratique, trouver un jeu de test idéal semble être une tâche ardue aussi nous nous contenterons de chercher des jeux de test ayant des propriétés intéressantes que nous détaillons dans la suite. Il est important de noter que le jeu de test devra être de taille modeste afin de garder un processus de test prenant un temps fini et raisonnable.

3.4 Critères de Test

Les propriétés portant sur les jeux de test s'expriment généralement en termes de *critères de test*, qui sont simplement des procédés de sélection des jeux de test. Nous proposons ici un premier critère très simple :

Definition 8 (Toutes_les_contraintes) *Le critère Toutes_les_contraintes exige que chaque contrainte d'un modèle M ait contribué au moins une fois à la résolution du système de contraintes.*

Autrement dit, satisfaire le critère *Toutes_les_contraintes* demande à ce que chaque contrainte du modèle ait été postée au moins une fois dans le système de contraintes. Un second critère, plus difficile à satisfaire, peut être défini comme suit :

Definition 9 (Tous_les_reveils) *Le critère Tous_les_reveils exige que chaque contrainte d'un modèle M ait contribué au moins deux fois à la résolution du système de contraintes.*

Autrement dit, satisfaire le critère *Tous_les_reveils* demande à ce que chaque contrainte du modèle ait été postée au moins une fois dans le système de contraintes et que chaque contrainte ait été réveillée au moins une fois lors de la résolution.

Pour illustrer cette notion de critère de test, nous appliquons le critère *toutes_les_contraintes* pour générer un jeu de test dans le cas des règles de Golomb. On rappelle que $P_x(k)$ est représenté sur la partie droite de la Fig. 1. Nous recherchons des instances k pour lesquelles toute contrainte du CPUT $P_x(k)$ est postée au moins une fois. Il y a 9 contraintes dans le CPUT $P_x(k)$ et le paramètre k qui est l'ordre de la règle, prend des valeurs dans $\mathcal{K} = 1.. + \infty$.

Pour $k = 1$, seule la contrainte numérotée 3 est postée :

```
(3)    x[1]=0 ;
```

Ainsi, le critère *toutes_les_contraintes* n'est pas satisfait. Pour $k = 2$, les contraintes 1, 2, 3, 4, 6, sont postées :

```
(1)    x[1] < x[2];
(2)    d[ind12] = x[ind12.j] - x[ind12.i];
(3)    x[1] = 0;
(4)    x[2] >= 1;
(6)    x[2] <= x[2] - x[1];
```

Mais les contraintes 5, 7, 8 et 9 n'ont toujours pas été postées. Pour $k = 3$, les contraintes suivante sont postées :

```
(1)    x[1] < x[2];
       x[2] < x[3];
(2)    d[ind12] = x[ind12.j] - x[ind12.i];
       d[ind13] = x[ind13.j] - x[ind13.i];
       d[ind23] = x[ind23.j] - x[ind23.i];
(3)    x[1] = 0;
(4)    x[3] >= 3;
(5)    allDifferent(d[ind12],d[ind13],d[ind23]);
(6)    x[2] <= x[3] - x[2];
(7)    d[ind13] = d[ind12] + d[ind23];
(9)    x[2] = x[1] + 1 => x[3] != x[2] + 1;
       x[2] = x[1] + 2 => x[3] != x[2] + 2;
       x[2] = x[1] + 3 => x[3] != x[2] + 3;
```

Mais, la huitième contrainte n'a toujours pas été postée. Enfin, avec $k = 4$, on a :

```
...
(8)    d[ind14] = d[ind13] + d[ind24] - d[ind23];
...
```

Ainsi, le critère *toutes_les_contraintes* est satisfait. Il est à noter que le jeu de test avec les instances $\{1, 2, 3, 4\}$ couvre bien le critère choisi mais que d'autres jeux de test auraient pu être choisis, comme par exemple $\{4\}$ ou $\{5, 6, 7\}$. Nous ne cherchons d'ailleurs pas à minimiser la taille du jeu de test puisque cela pourrait conduire à diminuer nos chances de détecter des non-conformités. Un sujet intéressant serait de générer automatiquement les instances du problème qui garantissent la couverture des critères de test proposées. Il faut aussi noter que les langages de modélisation intègrent des instructions qui rendent le système de contraintes conditionnel. Par exemple, OPL propose une contrainte *if_then_else* qui peut être paramétrée par les valeurs des paramètres k.

Nous proposons ici d'autres critères de test pour les problèmes qui recherchent une seule solution (Fig.3). L'idée sous-jacente est de regarder la structure d'un modèle à contraintes dérivé du problème de conformité. En effet, la recherche de non-conformités peut être adressée avec la remarque suivante : Si $\exists k, sol(P_x(k) \wedge \neg M_x(k)) \neq \emptyset$ alors $P_x(k)$ est non-conforme à $M_x(k)$. Sachant que $M_x(k) \equiv (C_1 \wedge C_2 ... \wedge C_n)$, on a :

$P_x(k) \wedge \neg M_x(k) \equiv$
$(P_x(k) \wedge \neg C_1) \vee (P_x(k) \wedge \neg C_2) \vee ..(P_x(k) \wedge \neg C_n)$
Rechercher des non-conformités peut être fait en résolvant un ou plusieurs de ces disjonctions. On a donc les critères suivants :

Definition 10 (Une_contrainte_niée) *Le critère Une_contrainte_niée exige que $\exists i$ tel que $sol(P_x(k) \wedge \neg C_i) \neq \emptyset$.*

Definition 11 (Une_paire_de_contraintes_niées) *Le critère Une_paire_de_contraintes_niées exige que $\exists i$ et j tels que $sol(P_x(k) \wedge \neg C_i) \neq \emptyset \wedge sol(P_x(k) \wedge \neg C_j) \neq \emptyset$.*

et ainsi de suite.

Illustrons le premier critère sur un exemple contenant une non-conformité.

Exemple : Soit X un ensemble de variables de type entier, N est un entier positif et val une valeur entière. Nous avons le modèle de départ $M_x(k)$ et le CPUT $P_x(k)$ et nous nous intéressons à la recherche d'une seule solution :

$M_x(k)$:	$P_x(k)$:
$\exists R \subseteq X,\ \|R\| = N$: $(\forall x \in R :\ x = val) \wedge (\forall y \in X \backslash R :\ y \neq val)$	$atMost(N, X, val)$

La contrainte globale $atMost(N, X, val)$ est vraie ssi au plus N variables de X valent val. En fait, nous faisons l'hypothèse qu'un développeur a raffiné le modèle $M_x(k)$ en utilisant cette contrainte globale.

Prenons maintenant une instance du modèle où k_i vaut $(N = 2, val = 3)$:

$M_x(k \backslash k_i)$:
$(x = 3 \vee y = 3) \wedge (x = 3 \vee z = 3) \wedge (y = 3 \vee z = 3) \wedge$
$(x = 3 \vee y = 3 \vee z = 3) \wedge (x \neq 3 \vee y = 3 \vee z = 3) \wedge$
$(x = 3 \vee y \neq 3 \vee z = 3) \wedge (x = 3 \vee y = 3 \vee z \neq 3) \wedge$
$(x \neq 3 \vee y \neq 3 \vee z \neq 3)$
$P_x(k \backslash k_i)$: $atMost(2, \{x, y, z\}, 3)$

En observant que $M_x(k \backslash k_i)$ est une conjonction de 8 contraintes, notées $C_1, .., C_8$ et si on cherche à générer un jeu de test pour le critère *Une_contrainte_niée* avec le disjonct C_2, alors :

on recherche les solutions de $atMost(2, \{x, y, z\}, 3) \wedge \neg C_2$ où $\neg C_2 \equiv (y \neq 3 \wedge z \neq 3)$. Ici, on trouve par exemple le point $(x, y, z) = (3, 9, 84)$ qui témoigne d'une non-conformité. Ainsi, cette génération de test démontre la non-conformité du CPUT par rapport à son modèle-oracle. Mais, un autre disjonct aurait pu être sélectionné et la non-conformité aurait pu rester non detectée (par exemple le disjonct $atMost(2, \{x, y, z\}, 3) \wedge \neg C_8$).

En effet, le CPUT qui utilise la contrainte globale $atMost$ est non-conforme au modèle-oracle initial car il implémente en fait un autre modèle, le modèle $M'_x(k)$ suivant :

$$\exists R \subseteq X,\ |R| = N\ :$$
$$(\forall x \in R :\ x = val) \Rightarrow (\forall y \in X \backslash R :\ y \neq val).$$

Il est important de noter que pour ces derniers critères de test, nous utilisons la négation des contraintes. Or, cette négation n'est pas toujours simple à déterminer ou calculer. Considérez par exemple, la négation d'une contrainte globale telle que $golomb(Var)$. Ce point est limitatif et nécessite de plus ample développements dans notre approche.

4 Validation

Dans cette section, nous proposons une première validation expérimentale de la Théorie proposée. Nous cherchons à vérifier que les notions introduites sont compatibles avec le développement de problèmes issus de l'Industrie. Nous utilisons un exemple dérivé d'un problème réel, celui de l'ordonnancement des véhicules (POV) de la distribution OPL. Nous utilisons les techniques de mutation ("mutation testing" [2]) qui consistent à injecter des fautes dans le CPUT afin de valider la qualité d'un jeu de test. Ces insertions systématiques de fautes sont souvent utilisées pour la vérification expérimentale de techniques de test.

Le problème d'ordonnancement des véhicules [10] illustre plusieurs caractéristiques intéressantes de la PPC incluant le paramétrage étendu du modèle, l'utilisation de contraintes redondantes et globales, l'utilisation de structures de données spécialisées.

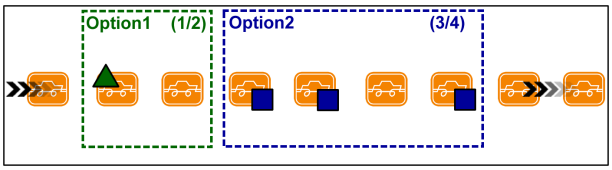


FIG. 7 – Chaine de montage.

Les contraintes de capacité qu'on trouve dans POV, sont formalisées en utilisant des contraintes de la forme " pas plus de r sur s " (r *outof* s), indiquant que chaque sous-séquence de s voitures, l'unité peut produire au maximum r voitures avec l'option en question (Fig. 7). Le problème d'ordonnancement des véhicules revient alors à trouver une affectation des voitures aux positions qui satisfait les contraintes de capacité. L'espace de recherche dans ce problème est composé des valeurs possibles des positions dans la chaîne de montage. Les voitures exigeant les mêmes options sont groupées dans une même classe. Dans ce qui suit, nous abuserons de la terminologie et nous référons simplement une voiture à une classe de voitures.

La partie gauche de la Fig. 8 donne le modèle–oracle de POV en OPL. Les variables *slot* sont des variables de décision (chaine de montage).

Le jeu de contraintes suivant représente les contraintes de capacité, qui sont exprimées en termes des variables d'installation des options($option[o][slot[s]]$). Si une contrainte de capacité pour une option o est de la forme " pas plus de l sur u ", on considère tous les sous-séquences de taille u de la chaîne de montage où il est nécessaire d'avoir au plus l éléments de la sous-séquence qui exigent l'option o. Ceci est exprimé dans OPL comme suit :

```
forall(o in Options & s in [1..nbSlots-capacity[o].u+1])
     sum(j in [s..s+capacity[o].u-1])
        option[o][slot[j]] <= capacity[o].l;
```

```
using CP;

int  nbCars    = ...; // # of cars
int   nbOptions = ...;// # of options
int   nbSlots   = ...;// # of slots

range   Cars    = 1..nbCars;
range   Options = 1..nbOptions;
range   Slots   = 1..nbSlots;

int demand[Cars] = ...;
int option[Options,Cars] = ...;

tuple Tcapacity {
    int l;
    int u;
};
Tcapacity capacity[Options] = ...;
int optionDemand[i in Options] = sum(j in Cars) demand[j] * option[i,j];
```

```
dvar int slot[Slots] in Cars;


subject to {
   // # of cars = demand
   forall(c in Cars )
      sum(s in Slots ) (slot[s] == c) == demand[c];

   forall(o in Options, s in 1..(nbSlots - capacity[o].u + 1) )
      sum(j in s..(s + capacity[o].u - 1))
            option[o][slot[j]] <= capacity[o].l;









};
```

- A -

```
dvar int slot[Slots] in Cars;
dvar int setup[Options,Slots] in 0..1;

subject to {
   // # of cars = demand
   forall(c in Cars )
      sum(s in Slots ) (slot[s] == c) == demand[c];

   forall(o in Options, s in 1..(nbSlots - capacity[o].u + 1) )
      sum(j in s..(s + capacity[o].u - 1))
            setup[o,j] <= capacity[o].l;

   forall(o in Options, s in Slots )
      setup[o,s] == option[o][slot[s]];

   forall(o in Options, i in 1..optionDemand[o])
      sum(s in 1 .. (nbSlots - i * capacity[o].u)) setup[o,s] >=
         optionDemand[o] - i * capacity[o].l;


};
```

- B -

FIG. 8 – $M_x(k)$ et $P_x(k)$ du POV en OPL.

L'efficacité de la résolution peut être améliorée en ajoutant une nouvelle structure de donnée `setup[o,s]`qui vaut 1 si l'option o est installée à la voiture qui occupe la position s, 0 sinon. A ce niveau, les variables `setup`et `slot`doivent être connectées :

```
forall(o in Options, s in Slots )
  setup[o,s] == option[o][slot[s]];
```

Il faut maintenant remplacer les `option[o,slot[s]]`par `setup[o,s]`. Ceci améliore nettement l'exécution du modèle. Puisque la résolution de contrainte sur des tableaux indexés de variable est couteuse. Il est ainsi préférable de factoriser ces expressions autant que possible.

Les contraintes redondantes n'enlèvent pas de solutions : elles expriment plutôt des propriétés sur les solutions qui peuvent aider la résolution à explorer l'espace de recherche plus efficacement. Un CPUT intermédiaire $P_x(k)$ peut être défini en ajoutant une contrainte redondante. Si l'option o a une contrainte de capacité "pas plus de l sur u", il s'ensuit que les dernières positions u peuvent contenir seulement l voitures demandant l'option o, donc les autres positions doivent contenir toutes les voitures restantes exigeant l'option o :

```
install[o,1] + ... +
install[o,nbPositions - u] >= demandeOption[o] - 1.
```

En règle générale, les dernières $k \times u$ positions peuvent seulement contenir $k \times l$ voitures exigeant l'option o.

```
install[o,1] + ... +
install[o,nbPositions - k*u] >= demandeOption[o] - k*l.
```

Ces contraintes sont définies dans OPL comme suit :

```
forall(o in Options & i in [1..DemandeOption[o]])
     sum(s in [1..nbPositions-i * capacite[o].u])
        install[o,s] >= DemandeOption[o] -i*capacite[o].l;
```

Ces raffinements nous amènent à au CPUT de la partie droite de la fig.8.

Si par erreur, dans cette contrainte redondante on a mis un " $<$ " au lieu du "$\geq$", le modèle intermediaire $P_x(k)$ na pas de solutions dans ce cas, on pourra dire à ce moment que $P_x(k)$ est non-conforme.
Nous avons créé 2 mutants du CPUT de POV, le tableau suivant montre les points de mutation dans le programme :

	Mutation
M1	3ème contrainte : s in Slots en s in Cars
M2	4ème contrainte : capacity[o].l en capacity[o].u

Le critère de couverture impose :

```
nbCars * nbOptions * nbSlots >= 1 ;
forall( o in Options) capacity[o].l <= capacity[o].u <= nbSlots ;
```

Nous prenons l'instance k_i qui satisfait le critère de couverture :

```
nbCars = 6;
nbOptions = 5;
```

```
nbSlots = 10;
demand = [1, 1, 2, 2, 2, 2];
option = [
          [ 1, 0, 0, 0, 1, 1],
          [ 0, 0, 1, 1, 0, 1],
          [ 1, 0, 0, 0, 1, 0],
          [ 1, 1, 0, 1, 0, 0],
          [ 0, 0, 1, 0, 0, 0]
       ];
capacity = [<1,2>,<2,3>,<1,3>,<2,5>,<1,5>];
```

Nous appliquons par la suite le critère Une_contrainte_niée en choisissant pour chaque mutant une contrainte à nier. Nous répétons ce processus encore une fois pour chaque mutant. Le tableau suivant donne l'ensemble des points de non-conformités atteints. **CC op.1** signifie : contrainte de capacité de l'option 1. On note s et non-s respectivement pour satisfaite et non-satisfaite :

	M1	**M1**
non-conformité	4 5 2 6 3 1 5 4 3 6	pas de solution
CC op.1	non-s	—
CC op.2	non-s	—
CC op.3	non-s	—
CC op.4	s	—
CC op.5	non-s	—

Le mutant 1 a été tué et on peut conclure qu'il n'est pas conforme au modèle original. Le mutant 2 n'a pas de solution ce qui nous permet de dire qu'il n'est pas conforme au modèle original.

5 Conclusion

Nous avons présenté les bases d'une première théorie du test pour les programmes à contraintes. Nous avons vu que la preuve de conformité entre le modèle original et le modèle raffiné est un problème difficile. Justement, nous avons adopté une démarche par réfutation qui consiste à aborder la problématique de la génération des cas de test de non-conformité. Ces cas de non conformité permettent d'exhiber les solutions du programme qui ne respectent pas sa spécification. Nous avons illustré cette stratégie sur des exemples, notamment le problème significatif de l'ordonnancement des véhicules. Le schéma de validation de la théorie donne une idée sur la difficulté du choix du critère de test. Les perspectives de ce travail sont multiples : la définition d'autres critères de test, la proposition d'un processus de génération des données de test pertinentes, le traitement de la négation des contraintes, et finalement l'automatisation du processus de test.

Références

[1] R. Backofen and S. Will. A constraint-based approach to fast and exact structure prediction in three-dimensional protein models. *Constraints*, 11(1) :5–30, 2006.

[2] R.A. DeMillo, R.J. Lipton, and F.G. Sayward. Hints on Test Data Generation : Help for the Practicing Programmer. *IEEE Computer Magazine*, 11(4) :34–41, Apr. 1978.

[3] P. Deransart, M. V. Hermenegildo, and J. Maluszynski, editors. *Analysis and Visualization Tools for Constraint Programming, Constrain Debugging (DiSCiPl project)*, volume 1870 of *Lecture Notes in Computer Science*. Springer, 2000.

[4] F. Fages, S. Soliman, and R. Coolen. Clpgui : A generic graphical user interface for constraint logic programming. *Constraints*, 9(4) :241–262, 2004.

[5] A. M. Frisch, W. Harvey, C. Jefferson, B. Martínez-Hernández, and I. Miguel. Essence : A constraint language for specifying combinatorial problems. *Constraints*, 13(3) :268–306, 2008.

[6] J. B. Goodenough and Susan L. Gerhart. Toward a theory of test data selection. In *Proceedings of the international conference on Reliable software*, pages 493–510, New York, NY, USA, 1975. ACM.

[7] J. S. Gourlay. *Theory of testing computer programs*. PhD thesis, Ann Arbor, MI, USA, 1981.

[8] L. Langevine, P. Deransart, M. Ducassé, and E. Jahier. Prototyping clp(fd) tracers : a trace model and an experimental validation environment. In *WLPE*, 2001.

[9] K. Marriott, N. Nethercote, R. Rafeh, P. J. Stuckey, M. Garcia De La Banda, and M. Wallace. The design of the zinc modelling language. *Constraints*, 13(3) :229–267, 2008.

[10] B. D. Parrello, Waldo C. Kabat, and L. Wos. Job-shop scheduling using automated reasoning : a case study of the car-sequencing problem. *J. Autom. Reason.*, 2(1) :1–42, 1986.

[11] W. T. Rankin. Optimal golomb rulers : An exhaustive parallel search implementation. Master's thesis, Duke University, Durham, 1993.

[12] P. Van Hentenryck. *The OPL optimization programming language*. MIT Press, Cambridge, MA, USA, 1999.

[13] Pascal Van Hentenryck and Laurent Michel. *Constraint-Based Local Search*. The MIT Press, 2005.

[14] E. J. Weyuker and T. J. Ostrand. Theories of program testing and the application of revealing subdomains. *IEEE Trans. Softw. Eng.*, 6(3) :236–246, 1980.

CHR **modulaire avec ask et tell**

François Fages[1], **Thierry Martinez**[1], **Cleyton Rodrigues**[2]

[1] Équipe Projet Contraintes, INRIA Paris–Rocquencourt, France
[2] Universidade Federal de Pernambuco, Recife, Brazil
{francois.fages,thierry.martinez}@inria.fr, cleyton.rodrigues@gmail.com

Résumé

Dans ce papier, nous introduisons une version modulaire du langage *Constraint Handling Rules* (CHR), appelé CHRat pour CHR modulaire avec *ask* et *tell*. Toute contrainte définie dans un composant CHRat peut être réutilisée à la fois dans les règles et les gardes d'un autre composant CHRat pour définir de nouveaux solveurs de contraintes. Contrairement aux travaux précédents sur la modularité de CHR, notre approche est complètement générale. Elle ne repose pas sur une condition de dérivabilité automatique de la vérification de l'implication des gardes, mais sur une discipline de programmation qui invite à définir par des règles CHRat la vérification à la fois de la satisfiabilité (*tell*) et de l'implication (*ask*) pour chaque contrainte. Nous définissons les sémantiques opérationnelles et déclaratives de CHRat, décrivons une transformation des composants CHRat en programmes CHR classiques, et prouvons la préservation de la sémantique. Nous donnons ensuite des exemples de modularisation pour des solveurs de contraintes CHR classiques.

1 Introduction

Le langage *Constraint Handling Rules* a été introduit il y a près de vingt ans comme langage déclaratif pour définir des solveurs de contraintes par des règles de réécriture de multi-ensembles avec gardes, étant donné un système de contraintes noyau prédéfini [10]. Le paradigme de programmation de CHR permet l'implantation de systèmes de contraintes en déclarant des règles de réécriture gardées qui transforment un *store* de contraintes en une forme résolue afin d'en déterminer la satisfiabilité. La forme résolue, atteinte lorsqu'il n'y a plus de transformation applicable, est insatisfiable si elle contient la contrainte $\langle \text{false} \rangle$, et est opérationnellement satisfiable sinon. Une propriété importante, mais non obligatoire, de ces transformations est la *confluence* qui signifie que la forme résolue est toujours indépendante de l'ordre d'application des règles, et constitue ainsi une *forme normale* pour le store de contraintes initial [1].

Depuis lors, CHR a évolué vers un langage de programmation généraliste à base de règles [10] avec des extensions telles que la gestion de la disjonction [3] ou l'introduction de types [5]. Cependant, un des inconvénients majeurs de CHR est l'absence de *modularité*. Une fois qu'un système de contraintes est défini en CHR à partir des contraintes du noyau, ce système de contraintes ne peut pas être réutilisé dans un autre programme CHR en bénéficiant des contraintes ainsi définies comme autant de nouvelles contraintes noyau. La raison de cette difficulté est qu'un programme CHR définit un test pour la satisfiabilité mais il n'en définit pas pour l'implication, pourtant nécessaire pour vérifier les gardes.

Les approches précédentes vis-à-vis de ce problème ont étudié des conditions sous lesquelles il est possible de dériver automatiquement un test d'implication à partir d'un test de satisfiabilité, notamment des conditions reposant sur l'équivalence logique[17] :

$$\mathcal{X} \models c \to d \text{ si et seulement si } \mathcal{X} \models (c \wedge d) \leftrightarrow c$$

Dans ce papier, nous proposons une version *modulaire* de CHR appelée CHR avec *ask* et *tell*, et notée CHRat [1]. Ce paradigme s'inspire du paradigme de programmation concurrente par contraintes [16, 15]. La discipline de programmation proposée dans CHRat pour la programmation modulaire de solveurs de contraintes demande, pour chaque contrainte introduite par le programmeur, de définir de règles de simplification et de propagation qui réécrivent des jetons de contrôle (*control token*) *ask* en des jetons de

[1] L'implantation de CHRat et des exemples de définition de hiérarchies de solveurs de contraintes sont disponibles à `http://contraintes.inria.fr/~tmartine/chrat`

contrôle *entailed*. La nécessité de définir des solveurs pour les *asks* et *tells* est déjà établie pour les implantations de systèmes de contraintes noyau [8] ; la discipline que nous proposons consiste à étendre ce principe aux solveurs CHR eux-mêmes. Une contrainte de garde $c(\vec{v})$ dans une instance de règle `R` est *opérationnellement impliquée* dans un store de contraintes contenant le jeton de contrôle `ask(K,` $c(\vec{v})$`)` lorsque sa forme résolue contient le jeton `entailed(K,`$c(\vec{v})$`)`, où `K` est une variable fraîche utilisée pour associer les jetons de contrôle à la règle `R`. Ceci nous permet de programmer des vérifications d'implications arbitrairement complexes par des règles, alors que ces vérifications reposent jusqu'à présent sur des programmes impératifs événementiels [7]. Avec cette discipline de programmation, les contraintes CHRat peuvent être réutilisées à la fois dans les règles et les gardes d'autres composants pour définir de nouveaux solveurs.

Dans la prochaine section, nous commençons par illustrer cette approche par un exemple simple. La section 3 énonce un certain nombre de définitions formelles sur CHR présenté comme un langage de réécriture de multi-ensembles, avec sa sémantique déclarative en logique linéaire (et en logique classique dans le cas de la réécriture ensembliste). Ensuite la section 4 introduit les sémantiques opérationnelles et déclaratives pour CHRat. La section 5 décrit la transformation de programmes CHRat en programmes CHR classiques et prouve sa correction. La section 6 illustre ensuite par des exemples la définition hiérarchique de solveurs de contraintes complexes. Enfin, nous concluons par une discussion sur la simplicité et l'expressivité de cette approche et des quelques limitations que nous rencontrons pour le moment.

2 Exemple introductif

2.1 Composants CHRat pour `leq/2` et `min/3`

Nous considérons pour commencer le programme CHR classique définissant une contrainte qui décrit une relation d'ordre.

Exemple 1 *Le solveur de satisfiabilité est défini par les quatre règles ci-dessous. Les trois premières règles traduisent les axiomes des relations d'ordre, et la dernière élimine les contraintes* `leq` *en double.*

```
leq(X,X) <=> true.
leq(X,Y), leq(Y,X) <=> X = Y.
leq(X,Y), leq(Y,Z) ==> leq(X,Z).
leq(X,Y) \ leq(X,Y) <=> true.
```

En CHRat, un solveur de contraintes doit définir des règles pour vérifier l'implication. Pour `leq(X,Y)`, ces règles réécrivent le jeton `ask(K,leq(X,Y))` en `entailed(K)`. `K` est une variable qui associe les jetons de contrôle aux instances des règles qui les ont générés, de façon à empêcher d'autres règles partageant la même garde de capturer ces jetons. Dans cet exemple, puisque le store est transitivement clos, l'implication de `leq(X,Y)` est directement observable dans le store par une simple règle si $X \neq Y$. Le cas réflexif est géré par une règle supplémentaire.

```
leq(X,Y)\ask(K,leq(X,Y)) <=>
    entailed(K,leq(X,Y)).
ask(K,leq(X,X)) <=>
    entailed(K,leq(X,X)).
```

Le solveur de satisfiabilité et le solveur d'implication pris ensembles définissent un composant CHRat pour la contrainte `leq(X,Y)`. Pour séparer les espaces de noms des modules CHRat, l'implantation repose sur un simple mécanisme de séparation de noms par différentiation des atomes (*atom-based*) [14] : les contraintes CHR exportées sont préfixées par le nom du composant et les contraintes CHR internes sont préfixées de façon à éviter les collisions [14]. De tels composant peuvent être réutilisés pour définir de nouveaux solveurs en utilisant la contrainte `leq(X,Y)` à la fois dans les règles et les gardes.

Exemple 2 *Le composant ci-dessous définit un solveur pour la contrainte minimum* `min(X,Y,Z)`*, établissant que* `Z` *est la valeur minimale parmi* `X` *et* `Y` *:*

```
component min_solver.
import leq/2 from leq_solver.
export min/3.
min(X,Y,Z) <=> leq(X,Y) | Z=X.
min(X,Y,Z) <=> leq(Y,X) | Z=Y.
min(X,Y,Z) ==> leq(Z,X), leq(Z,Y).

min(X,Y,Z) \ ask(K, min(X,Y,Z)) <=>
    entailed(K, min(X,Y,Z)).
ask(K, min(X, Y, X)) <=> leq(X, Y) |
    entailed(K, min(X,Y,Z)).
ask(K, min(X, Y, Y)) <=> leq(Y, X) |
    entailed(K, min(X,Y,Z)).
```

2.2 Transformation vers un programme CHR classique

Dans CHR, les gardes sont restreintes aux contraintes du noyau [10]. Les composants CHRat sont transformés en programmes CHR classiques par une transformation de programme qui :

- retire des gardes toutes les contraintes définies par l'utilisateur ;
- renomme les prédicats `ask(K,c(...))` en `ask_c(K,...)` avec un argument supplémentaire pour la variable d'association `K` ;

– introduit pour chaque règle CHRat une contrainte CHR id_n qui relie la variable d'association aux instances des variables de la tête ;
– sépare les espaces de noms.

Par exemple, le solveur CHRat `min` est transformé en le programme CHR suivant (modulo séparation de l'espace de noms par soucis de concision) :

```
min(X,Y,Z)\ask_min(K,X,Y,Z)==>
    entailed_min(K,X,Y,Z).
min(X,Y,Z)==>ask_leq(K,X,Y),id1(K,X,Y,Z).
id1(X,Y,Z,K),min(X,Y,Z),
    entailed_leq(K,X,Y)<=>Z=X.
min(X,Y,Z)==>ask_leq(K,Y,X),id2(K,X,Y,Z).
id2(X,Y,Z,K),min(X,Y,Z),
    entailed_leq(K,Y,X)<=>Z=Y.
min(X,Y,Z)==>leq(Z,X),leq(Z,Y).
ask_min(X,Y,X,K)==>
    ask_leq(K0,X,Y),id3(K0,X,Y,K).
id3(K0,X,Y,K),ask_min(K,X,Y,X),
    entailed_leq(K0,X,Y)<=>
    entailed_min(K,X,Y,X).
ask_min(X,Y,Y,K)==>
    ask_leq(K0,Y,X),id4(K0,X,Y,K).
id4(K0,X,Y,K),ask_min(X,Y,Y,K),
    entailed_leq(K0,Y,X)<=>
    entailed_min(K,X,Y,Y).
```

2.3 Variables quantifiées existentiellement dans les gardes

En CHRat, comme en CHR, les variables qui apparaissent dans une garde sans apparaître dans la tête de la règle nécessitent un traitement spécifique. Par exemple, considérons la règle CHRat suivante pour éliminer les éléments non minimaux :

```
number(A), number(B) <=> min(A,B,C) |
    number(C).
```

Le solveur d'implication défini précédemment ne peut pas détecter l'implication d'une telle garde.

En CHRat, les variables `V` quantifiées existentiellement dans les gardes sont marquées par des jetons de contrôle `exists(K,V)`, avec `K` la variable d'association introduite par l'instance de la règle dont la garde est à vérifier. Les règles pour `min/2` dans l'exemple 2 sont ainsi complétées avec des règles supplémentaires pour instancier les variables quantifiées existentiellement de la manière suivante :

```
ask(K,min(X,Y,Z)),exists(K,Z) <=>
    leq(X,Y) |
    Z=X, entailed(K, min(X,Y,Z)).
ask(K,min(X,Y,Z)),exists(K,Z) <=>
    leq(Y,X) |
        Z=Y, entailed(K, min(X,Y,Z)).
min(X,Y,M) \ ask(K, min(X,Y,Z)),
    exists(K,Z) <=> leq(X,Y) |
        Z=M, entailed(K, min(X,Y,Z)).
```

Toutes ces règles ajoutent, en plus du jeton `entailed`, la contrainte elle-même, bien qu'elle soit déjà impliquée, dans le store, ceci afin de contraindre les variables existentiellement quantifiées de satisfaire la contrainte. Ceci généralise la solution proposée dans [1] pour la sémantique opérationnelle de CHR pour les contraintes noyau aux contraintes définies par l'utilisateur.

3 Préliminaires sur CHR

Notations

Pour toute fonction f et tout sous-ensemble $X \subseteq \mathrm{dom}(f)$, nous notons $f(X) = \{f(x) \mid x \in X\}$ et cette notation s'étend aux fonctions n-aires par produit cartésien. Pour tout ensemble E, un *multi-ensemble* M sur E est représenté par une fonction de multiplicité $C_M : E \to \mathbb{N}$. Le *support* de M est $\sup(M) = \{e \in E \mid C_M(e) \geq 1\}$. Un multi-ensemble est fini si son support est fini. Un multi-ensemble est (identifié à) un ensemble lorsque $\forall e \in E, C_M(e) \leq 1$. Soit M et M' deux multi-ensembles sur E, la *somme multi-ensembliste* $M \oplus M'$ est définie par $C_{M\oplus M'} : e \mapsto C_M(e) + C_{M'}(e)$ et la *différence multi-ensembliste* $M \ominus M'$ est définie par $C_{M\ominus M'} : e \mapsto \min(C_M(e) - C_{M'}(e), 0)$. $M \subseteq M'$ lorsque pour tout $e \in E$, $C_M(e) \leqslant C_{M'(e)}$. Une *fonction multi-ensembliste* f de M dans M' ([13]) est représentée par une fonction $\sigma_f : E \to E$ telle que $\sigma_f(\sup(M)) \subseteq \sup(M')$: pour tout $N \subseteq M$, $f(N)$ désigne le multi-ensemble défini par $C_{f(N)} = C_N \circ \sigma_f^{-1}$. f est une *injection* si σ_f est une permutation et $f(M) \subseteq M'$. Une fonction multi-ensembliste f' de $N \supseteq M$ dans M' est une *extension* de f si $\sigma_{f'}$ et σ_f coïncident sur $\sup(M)$.

Syntaxe

Les programmes CHR sont construits autour de trois signatures disjointes : une signature Σ pour les symboles de constantes et de fonctions, une signature Π pour les symboles de prédicats définis par l'utilisateur et une signature $\Pi_{\mathcal{X}}$ pour les symboles de prédicats des contraintes du noyau, $\Pi_{\mathcal{X}}$ est supposée contenir le prédicat d'égalité $=$ et les constantes true et false. Les contraintes définies par l'utilisateur sont les propositions atomiques $\mathcal{A}$ sur Π.

Le langage des contraintes noyau est un fragment $\mathcal{L}$ de Σ-$\Pi_{\mathcal{X}}$-formules logiques du premier ordre, clos par conjonction et quantification existentielle. Les contraintes noyau sont interprétées sur une structure fixée $\mathcal{X}$. Une contrainte $c \in \mathcal{L}$ est *$\mathcal{X}$-satisfiable* si $\mathcal{X} \models \exists \vec{x}(c)$ où $\vec{x}$ est l'ensemble des variables libres de c,

dénoté $V(c)$. Une contrainte c *implique* une contrainte d de $\mathcal{X}$ si $\mathcal{X} \models \forall \vec{x}(c \rightarrow d)$ où $\mathcal{X} \models \forall \vec{x}(c \rightarrow d)$ avec $\vec{x} = V(c) \cup V(d)$. Nous supposons que la satisfiabilité et l'implication sont décidables dans $\mathcal{X}$.

Un programme CHR($\mathcal{X}$) P est une suite finie de règles de réécriture de la forme suivante : $H \backslash H' \Leftrightarrow G \mid B$ où les *têtes* H et H' sont des multi-ensembles finis de contraintes utilisateur, la *garde* G est une conjonction de contraintes noyau, et le *corps* B est la paire $(B_{\mathcal{X}}, B_u)$ où $B_{\mathcal{X}}$ est une conjonction de contraintes noyau et B_u est un multi-ensemble de contraintes utilisateur. Une *règle de simplification* est une règle où H' est vide, et est notée $H' \Leftrightarrow G \mid B$. Une *règle de propagation* est une règle où H est vide, et est notée $H \Rightarrow G \mid B$. Une *règle de simpagation* est une règle où H et H' sont tous deux non-vides. H et H' ne peuvent pas être simultanément vides.

Sémantique opérationnelle

Soit $\rightarrow$ la relation de transition sur les *états* (T, c) avec un store contenant le multi-ensemble des contraintes définies par l'utilisateur T et une contrainte noyau c. Une règle s'applique à un état (T, c) si sa tête correspond à un multi-ensemble de T et si sa garde est impliquée par c. Formellement :

$$(T, c) \rightarrow (\langle T \ominus \pi(H') \oplus B_u \rangle, \langle c \wedge c_m \wedge B_{\mathcal{X}} \rangle)$$

s'il existe une règle $H \backslash H' \Leftrightarrow G \mid B_{\mathcal{X}}, B_u$ dans P (avec les variables renommées en dehors de (T, c)) et une injection π de $T' = H \oplus H'$ dans T telle que la contrainte $c_m = \langle (\bigwedge_{t \in \sup(T')} t = \sigma_\pi(t)) \wedge G \rangle$, dite de correspondance (*matching*) soit impliquée, c'est-à-dire $\mathcal{X} \models \forall \vec{x}(B_{\mathcal{X}} \rightarrow \exists \vec{y}(c_m))$ où $\vec{x} = V(B_{\mathcal{X}})$ et $\vec{y} = V(c_m) \setminus V(c)$.

En considérant les dérivations à partir d'un état initial (T_0, c_0), aussi appelée *requête*, l'observation de tous les états accessibles nous conduit à considérer la sémantique opérationnelle suivante :

$$\mathcal{O}^a_P(T_0, c_0) = \{(T, c) \mid (T_0, c_0) \xrightarrow{*} (T, c)\}$$

avec $\xrightarrow{*}$ désignant la clôture réflexive et transitive de $\rightarrow$, et $\not\rightarrow$ le fait qu'aucune transition ne s'applique.

Sémantiques déclaratives

CHR possède deux sémantiques déclaratives : l'une en logique classique [10] pour les programmes linéaires à gauche et l'une en logique linéaire sans restriction [4]. Une règle est dite *linéaire à gauche* si toute contrainte utilisateur ne correspond qu'à au plus une contrainte dans la tête de cette règle. Un programme est linéaire à gauche dès lors que toutes les règles le sont.

En logique classique, un multi-ensemble M de contraintes est interprété par la conjonction $M^\dagger = \langle \bigwedge_{c \in \sup(M)} c \rangle$ de ses éléments. Pour chaque règle CHR :

$$\begin{aligned}&\left(H \backslash H' \Rightarrow G \mid B_{\mathcal{X}}, B_u\right)^\dagger = \\ &\quad \forall \vec{x}(\exists \vec{y}(G) \wedge H^\dagger \rightarrow (H'^\dagger \leftrightarrow \exists \vec{z}(G \wedge B_{\mathcal{X}} \wedge B_u^\dagger)))\end{aligned}$$

où $\vec{x} = V(H_0, H_1)$, $\vec{y} = V(G) \setminus V(H_0, H_1)$ et $\vec{z} = V(G, B_{\mathcal{X}}, B_u) \setminus V(H_0, H_1)$. L'interprétation logique d'un programme $(P)^\dagger$ est la conjonction logique des interprétations de chacune des règles de P. La sémantique logique de la requête q est :

$$\mathcal{L}_P(q) = \{c \mid (P)^\dagger \models_{\mathcal{X}} q \rightarrow c\}$$

$\mathcal{L}_P(q)$ est clos par implication logique dans cette traduction. Pour tout ensemble S de fait logique, nous notons $\downarrow S = \{c' \mid \exists c \in S, \models_{\mathcal{X}} c \rightarrow c'\}$. Dès lors, $\mathcal{L}_P(q) = \downarrow \mathcal{L}_P(q)$.

Soit V un ensemble de variables libres sur la requête (T_0, c_0). Les états sont interprétés comme $(T, c)^\dagger = \langle \exists \vec{x}((T)^\dagger \wedge c) \rangle$ où $\vec{x} = V(T, c) \setminus V(T_0, c_0)$: les variables de la requête apparaissent dans les observables et sont donc laissées libres dans l'interprétation. La notation est étendue aux ensembles S d'états : $S^\dagger = \{(T, c)^\dagger \mid (T, c) \in S\}$.

Théorème 1 *Pour tout programme* CHR($\mathcal{X}$) *linéaire à gauche P et toute requête (T_0, c_0) :*

$$\downarrow (\mathcal{O}^a_P(T_0, c_0))^\dagger \subseteq \mathcal{L}_P((T_0, c_0)^\dagger)$$

Cette formulation donne seulement un résultat de correction : $\mathcal{L}_P((T_0, c_0)^\dagger) \subseteq \downarrow (\mathcal{O}^a_P(T_0, c_0))^\dagger$ n'est pas vrai en général. Par exemple, considérons le programme $\{\texttt{a <=> b. a <=> c.}\}$: $\mathcal{L}_P((\texttt{a}, \top)^\dagger) = \downarrow\{\texttt{a} \wedge \texttt{b} \wedge \texttt{c}\}$ alors que $\downarrow (\mathcal{O}^a_P(\texttt{a}, \top))^\dagger = \{\texttt{a}, \texttt{b}, \texttt{c}, \top\}$. Un résultat de complétude plus faible est vérifié : $\mathcal{L}_P((T_0, c_0)^\dagger) \subseteq \mathcal{L}_P((\mathcal{O}^a_P(T_0, c_0))^\dagger)$: dans ce cas, la propriété est une conséquence directe de $(T_0, c_0)^\dagger \in \mathcal{L}_P((\mathcal{O}^a_P(T_0, c_0))^\dagger)$.

La sémantique classique n'est pas correcte si P n'est pas linéaire à gauche. Par exemple, avec la simple règle `a, a <=> b`, nous avons $\mathcal{L}_P((\texttt{a}, \top)^\dagger) = \downarrow\{\texttt{a} \wedge \texttt{b}\}$ tandis que $\downarrow (\mathcal{O}^a_P(\texttt{a}, \top))^\dagger = \downarrow\{\texttt{a}\} \neq \downarrow\{\texttt{a} \wedge \texttt{b}\}$. De plus, cette sémantique logique identifie trop de programmes (par exemple, $\{\texttt{a <=> b.a <=> c.}\}$ et $\{\texttt{a <=> b, c.}\}$).

Pour pallier à ces limitations, une sémantique déclarative exprimée dans la logique linéaire de Girard [12] a été développée [4] pour tout programme CHR. Les contraintes noyau sur $\mathcal{X}$ sont traduits par la traduction de Girard en logique linéaire en faisant usage de l'opérateur *bien sûr* noté ! [12]. Un multi-ensemble M de contraintes est interprété par la conjonction multiplicative des contraintes $M^{\dagger\dagger} = \langle \bigotimes_{c \in M} c \rangle$. Pour toute

règle CHR :

$$\big(H \backslash H' \Rightarrow G \mid B\big)^{\dagger\dagger} =$$
$$\langle !\forall \vec{x}(\exists \vec{y}(G^{\dagger\dagger}) \otimes H^{\dagger\dagger} \otimes H'^{\dagger\dagger} \multimap \exists \vec{z}(H^{\dagger\dagger} \otimes G^{\dagger\dagger} \otimes B_{\mathcal{X}}^{\dagger\dagger} \otimes B_u^{\dagger\dagger}))\rangle$$

où $\vec{x} = V(H, H')$, $\vec{y} = V(G) \setminus V(H, H')$ et $\vec{z} = V(G, B_{\mathcal{X}}, B_u) \setminus V(H, H')$.

L'interprétation en logique linéaire d'un programme $(P)^{\dagger}$ est la conjonction multiplicative des interprétations des règles de P. La sémantique en logique linéaire d'une requête q est :

$$\mathcal{LL}_P(q) = \{c \mid (P)^{\dagger\dagger} \models_{LL,\mathcal{X}} q^{\dagger\dagger} \multimap c \otimes \top\}$$

$\mathcal{LL}_P(q)$ est clos par implication linéaire. Pour tout ensemble S de faits logiques, nous notons $\downarrow S = \{c' \mid \exists c \in S, \models_{LL,\mathcal{X}} c \multimap c' \otimes \top\}$. Dès lors, $\mathcal{LL}_P(q) = \downarrow \mathcal{LL}_P(q)$.

Soit V l'ensemble des variables libres de la requête (T_0, c_0). Les états sont interprétés comme $(T, c)^{\dagger\dagger} = \langle \exists \vec{x}(T^{\dagger\dagger} \otimes c^{\dagger\dagger})\rangle$ où $\vec{x} = V(T, c) \setminus V(T_0, c_0)$: les variables de la requête apparaissent dans les observables et sont donc laissées libres dans l'interprétation. La notation est étendue aux ensembles S d'états : $S^{\dagger\dagger} = \{(T, c)^{\dagger\dagger} \mid (T, c) \in S\}$.

Théorème 2 ([4, 9]) *Pour tout programme* CHR($\mathcal{X}$) P *et requête* (T_0, c_0) *:*

$$\downarrow (\mathcal{O}_P^a(T_0, c_0))^{\dagger\dagger} = \mathcal{LL}_P((T_0, c_0)^{\dagger\dagger})$$

4 Syntaxe et sémantique des composants CHRat($\mathcal{X}$)

Syntaxe

En CHRat, les jetons de contrôle sont construits sur des symboles frais issus de la signature $\Pi_t = \{\text{ask}/2, \text{exists}/2, \text{entailed}/2\}$, disjointe de Π et $\Pi_{\mathcal{X}}$.

Définition 1 *Un* programme CHRat($\mathcal{X}$) P *est une suite finie de règles* $H \backslash H' \Leftrightarrow G \mid B_{\mathcal{X}}, B_u$ *où* H *est un multi-ensemble d'éléments de* $\mathcal{A}$ *;* H' *est un multi-ensemble d'éléments de* $\mathcal{A} \cup \text{ask}(V, \mathcal{A}) \cup \text{exists}(V, V)$ *;* G *est une formule de* $\mathcal{L}'$ *où* $\mathcal{L}'$ *est l'extension de* $\mathcal{L}$ *contenant les propositions atomiques* $\mathcal{A}$ *et close par conjonction et quantification existentielle ;* $B_{\mathcal{X}}$ *est une formule de* $\mathcal{L}$ *;* B_u *est un multi-ensemble d'éléments de* $\mathcal{A} \cup \text{entailed}(V, \mathcal{A})$*.* H *et* H' *ne peuvent pas être simultanément vides.*

En tant que formule de $\mathcal{L}'$, une garde G est de la forme $G_u \wedge G_{\mathcal{X}}$ où G_u est une conjonction (ou multi-ensemble) de jetons et $G_{\mathcal{X}}$ est une formule de $\mathcal{L}$. L'ensemble des variables qui n'apparaissent que dans G_u est noté $V_G = V(G_u) \setminus V(H, H', G_{\mathcal{X}})$.

Sémantique opérationnelle

Les états sont des triplets (T, c, I), où :
- le store noyau c est une contrainte de $\mathcal{C}$;
- le store utilisateur T est un multi-ensemble sur $\mathcal{A} \cup \text{ask}(V, \mathcal{A}) \cup \text{exists}(V, V) \cup \text{entailed}(V, \mathcal{A})$;
- la table des instances en attente I est une application à support finie des variables vers les instances de règles $(\mathbf{r}, \sigma_\pi)$ où $\mathbf{r}$ est une règle (renommée) et σ_π une permutation de $\mathcal{A}$.

Pour toute table d'instances en attente I, $\sup(I)$ désigne le support de I. Si $K \in V$ est tel que $K \notin \sup(I)$, alors $I \uplus (\mathbf{r}, \sigma_\pi)_K$ désigne l'application I' telle que $\sup(I') = \sup(I) \cup \{K\}$ avec $I'|_{\sup(I)} = I$ et $I'(K) = (\mathbf{r}, \sigma_\pi)$.

Nous introduisons deux types de transitions déclenchables depuis un état (T, c, I) : les suspensions $\xrightarrow{\text{s}}$ et les réveils $\xrightarrow{\text{w}}$.

– la règle de suspension :

$$(T, c, I) \xrightarrow{\text{s}} (\langle T \oplus \text{ask}(K, G_u) \oplus \text{exists}(K, V_G)\rangle, \langle c \wedge \exists(c_m)\rangle, \langle I \uplus (\mathbf{r}, \sigma_\pi)_K\rangle)$$

où $\mathbf{r} = \langle H \backslash H' \Rightarrow G \mid B_{\mathcal{X}}, B_u\rangle$ est une règle de P renommée avec des variables fraîches par rapport à (T, c, I) et π une injection de $T' = H \oplus H'$ dans T telle que la contrainte de correspondance $c_m = \langle(\bigwedge_{t \in \sup(T')} t = \sigma_\pi(t)) \wedge G_{\mathcal{X}}\rangle$ soit impliquée, c'est-à-dire $\mathcal{X} \models \forall \vec{x}(c \rightarrow \exists \vec{y}(c_m))$ où $\vec{x} = V(c)$ et $\vec{y} = V(c_m) \setminus V(c)$. Alors la transition ajoute la nouvelle instance en attente $(\mathbf{r}, \sigma_\pi)$ à I, indexée par une variable fraîche K.

– la règle de réveil :

$$(T, c, \langle I \uplus (\mathbf{r}, \sigma_\pi)_K\rangle) \xrightarrow{\text{w}} (\langle T \ominus \pi'(H') \ominus \pi'(\text{entailed}(K, G_u)) \oplus B_u\rangle, \langle c \wedge c_m \wedge B_{\mathcal{X}}\rangle, I)$$

où $(\mathbf{r}, \sigma_\pi)$ est une instance en attente de la règle $\mathbf{r} = \langle H \backslash H' \Rightarrow G \mid B_{\mathcal{X}} \wedge B_u\rangle$ et σ_π représente une injection π de $H \oplus H'$ dans T, telle qu'il existe une injection π' de $T' = H \oplus H' \oplus \text{entailed}(K, G_u)$ dans T, qui étende π de telle sorte que la contrainte de correspondance : $c_m = \langle(\bigwedge_{t \in \sup(T')} t = \sigma_{\pi'}(t)) \wedge G_{\mathcal{X}}\rangle$ soit impliquée, c'est-à-dire $\mathcal{X} \models \forall \vec{x}(c \rightarrow \exists \vec{y}(c_m))$ où $\vec{x} = V(c)$ et $\vec{y} = V(c_m) \setminus V(c)$.

Nous définissons $\rightarrow = \xrightarrow{\text{s}} \cup \xrightarrow{\text{w}}$.

Définition 2 *La* sémantique opérationnelle *pour l'observation des états accessibles depuis une requête* $q = T_0 \wedge c_0 \in \mathcal{L}'$ *est, avec* $\vec{x} = V(T_1, c) \setminus V(q)$ *:*

$$\mathcal{O}_P^a[q] = \{\exists \vec{x}(T_1 \wedge c) \mid (T_0, c_0, \emptyset) \xrightarrow{*} (T, c, \_), T_1 = T \cap \mathcal{A}\}$$

La sémantique des stores accessibles $\mathcal{O}^a_P[q]$ est reliée aux sémantiques déclaratives de la prochaine section. Les stores observés sont restreints à $\mathcal{A}$. Soit min le solveur défini dans l'exemple 2, alors $\langle \text{leq}(X,Y) \wedge Z = X \rangle$ est un store accessible et le jeton $\text{ask}(\text{leq}(Y,X))$ introduit dans l'infructueux test d'implication (pour la seconde règle du composant leq) n'est pas exposé.

Sémantique déclarative

Nous définissons la transformation $(\cdot)^*$ des jetons de contrôle et des instances suspendues aux propositions atomiques :

$$(\text{ask}(K, p(\overrightarrow{X})))^* = \text{ask}_p(K, \overrightarrow{X})$$
$$(\text{entailed}(K, p(\overrightarrow{X})))^* = \text{entailed}_p(K, \overrightarrow{X})$$
$$(\text{exists}(K, V))^* = \text{exists}(K, V)$$
$$(((H \backslash H' \Rightarrow G \mid B), \sigma_\pi)_K)^* = \text{id}_{\mathbf{r}}(K, \overrightarrow{V}(\sigma_\pi(H \oplus H')))$$

En logique classique, un multi-ensemble M de contraintes est interprété par la conjonction $M^\ddagger = \langle \bigwedge_{c \in \text{sup}(M)} c^* \rangle$ de ses éléments. Une règle est logiquement interprétée comme :

$$\big(H \backslash H' \Rightarrow G \mid B_\mathcal{X}, B_u\big)^\ddagger =$$
$$\big(H \backslash H' \Rightarrow G \mid B_\mathcal{X}, B_u\big)^\ddagger_{\text{s}} \wedge \big(H \backslash H' \Rightarrow G \mid B_\mathcal{X}, B_u\big)^\ddagger_{\text{w}}$$

où les sous-formules *suspend* et *wake* traduisent leur contrepartie opérationnelle :

$$\big(H \backslash H' \Rightarrow G \mid B_\mathcal{X}, B_u\big)^\ddagger_{\text{s}} = \Big\langle \forall \vec{x} \Big(\exists \vec{y}(G) \wedge H^\ddagger \wedge H'^\ddagger \rightarrow \exists K \big(\text{id}(K, \vec{z}) \wedge \text{exists}(K, V_G) \wedge \big(\bigwedge_{p(\vec{u}) \in \text{sup}(G_u)} \text{ask}_p(K, \vec{u})\big)\big)\Big)\Big\rangle$$

$$\big(H \backslash H' \Rightarrow G \mid B_\mathcal{X}, \wedge B_u\big)^\ddagger_{\text{w}} = \Big\langle \forall \vec{x} \forall K \big(\exists \vec{y}(G) \wedge H^\ddagger \rightarrow (H'^\ddagger \wedge \text{id}(K, \vec{z}) \wedge \big(\bigwedge_{p(\vec{u}) \in G_u} \text{entailed}_p(K, \vec{u})\big) \leftrightarrow \exists \vec{z}(G \wedge B_\mathcal{X} \wedge B^\ddagger_u))\big)\Big\rangle$$

avec $\vec{x} = V(H, H')$, $\vec{y} = V(G) \setminus V(H, H')$ et $\vec{z} = V(G, B_\mathcal{X}, B_u) \setminus V(H, H')$. L'interprétation logique d'un programme $(P)^\ddagger$ est la conjonction logique des interprétations des règles de P.

Définition 3 *La* sémantique en logique classique *d'une règle $q \in \mathcal{L}'$ est :*

$$\mathcal{L}_P[q] = \{c \in \mathcal{L}' \mid (P)^\ddagger \models_\mathcal{X} q \rightarrow c\}$$

Soit V l'ensemble des variables libres de la requête $q = T_0 \wedge c_0 \in \mathcal{L}'$. Les états sont interprétés comme $(T, c, I)^\ddagger = \langle \exists \vec{x}(T^\ddagger \wedge c \wedge (\bigwedge_{K \in \text{sup}(I)} I(K)^*)) \rangle$ où $\vec{x} = V(T, c) \setminus V(T_0, c_0)$: les variables de la requête apparaissent dans les observables et sont donc laissées libres dans l'interprétation. La définition est étendue pour un ensemble d'états S : $S^\ddagger = \{(T, c, I)^{\dagger\dagger} \mid (T, c, I) \in S\}$.

Avec la même restriction que pour CHR, la sémantique opérationnelle de CHRat est correcte vis-à-vis de la sémantique en logique classique. Ce résultat découle du lemme suivant :

Lemme 1 *Si $c_0 \rightarrow c_1$, alors $(P)^\ddagger \models_\mathcal{X} (c_0)^\ddagger \rightarrow (c_1)^\ddagger$.*

Théorème 3 *Pour tout programme* CHRat$(\mathcal{X})$ *linéaire à gauche P et toute requête $q \in \mathcal{L}'$:*

$$\downarrow \mathcal{O}^a_P[q] \subseteq \mathcal{L}_P[q]$$

La sémantique en logique linéaire utilise des propositions atomiques similaires pour coder les jetons de contrôle et les instances suspendues. Un multi-ensemble M de contraintes est interprété par la conjonction multiplicative des contraintes $M^{\ddagger\ddagger} = \langle \bigotimes_{c \in M} c^* \rangle$. Une règle est interprétée en logique linéaire de la manière suivante :

$$\big(H \backslash H' \Rightarrow G \mid B\big)^{\ddagger\ddagger} = \big(H \backslash H' \Rightarrow G \mid B\big)^{\ddagger\ddagger}_{\text{s}} \otimes \big(H \backslash H' \Rightarrow G \mid B\big)^{\ddagger\ddagger}_{\text{w}}$$

$$\big(H \backslash H' \Rightarrow G \mid B_\mathcal{X}, B_u\big)^{\ddagger\ddagger}_{\text{s}} = \Big\langle !\forall \vec{x} \Big(\exists \vec{y}(G^{\ddagger\ddagger}) \otimes H^{\ddagger\ddagger} \otimes H'^{\ddagger\ddagger} \multimap \exists K \big(H^{\ddagger\ddagger} \otimes H'^{\ddagger\ddagger} \otimes \text{id}(K, \vec{z}) \otimes \text{exists}(K, V_G) \otimes \big(\bigotimes_{p(\vec{u}) \in G_u} \text{ask}_p(K, \vec{u})\big)\big)\Big)\Big\rangle$$

$$\big(H \backslash H' \Rightarrow G \mid B_\mathcal{X}, B_u\big)^{\ddagger\ddagger}_{\text{w}} = \Big\langle !\forall \vec{x} \forall K \Big(\exists \vec{y}(G^{\ddagger\ddagger}) \otimes H^{\ddagger\ddagger} \otimes H'^{\ddagger\ddagger} \otimes \text{id}(K, \vec{z}) \otimes \big(\bigotimes_{p(\vec{u}) \in G_u} \text{entailed}_p(K, \vec{u})\big) \multimap \exists \vec{z}(H^{\ddagger\ddagger} G^{\dagger\dagger} \otimes B_\mathcal{X} \otimes B^{\ddagger\ddagger}_u)\big)\Big)\Big\rangle$$

avec $\vec{x} = V(H, H')$, $\vec{y} = V(G) \setminus V(H, H')$ et $\vec{z} = V(G, B_\mathcal{X}, B_u) \setminus V(H, H')$. : L'interprétation d'un programme $(P)^{\ddagger\ddagger}$ est la conjonction multiplicative des interprétations des règles de P.

Définition 4 *La* sémantique en logique linéaire *d'une requête $q \in \mathcal{L}'$ est*

$$\mathcal{LL}_P[q] = \{c \in \mathcal{L}' \mid (P)^{\ddagger\ddagger} \models_{LL,\mathcal{X}} q \multimap c \otimes \top\}$$

Soit V l'ensemble des variables libres de la requête $q = T_0 \wedge c_0 \in \mathcal{L}'$. Les états sont interprétés comme $(T, c, I)^{\ddagger\ddagger} = \langle \exists \vec{x} (T^{\ddagger\ddagger} \otimes c \otimes (\bigotimes_{K \in I} I(K)^*)) \rangle$ où $\vec{x} = V(T, c) \setminus V(T_0, c_0)$: les variables de la requête apparaissent dans les observables et sont donc laissées libres dans l'interprétation. Cette définition est étendue pour un ensemble d'états S : $S^{\ddagger\ddagger} = \{(T, c, I)^{\ddagger\ddagger} \mid (T, c, I) \in S\}$.

La sémantique opérationnelle de CHRat est correcte et complète vis-à-vis de la sémantique en logique linéaire. Ce résultat découle du lemme suivant :

Lemme 2 *Si* $c_0 \to c_1$, *alors* $(P)^{\ddagger\ddagger} \models_{LL,\mathcal{X}} (c_0)^{\ddagger\ddagger} \multimap (c_1)^{\ddagger\ddagger}$.

La correction et la complétude de la sémantique opérationnelle par rapport à la sémantique en logique linéaire est établie par le théorème suivant :

Théorème 4 *Pour tout programme* CHRat($\mathcal{X}$) P *et toute requête* $q \in \mathcal{L}'$:

$$\downarrow \mathcal{O}^a_P[q] = \mathcal{LL}_P[q]$$

5 Transformation de programme de CHRat vers CHR

Définition 5 *Soit* $[\![\cdot]\!]$: CHRat($\mathcal{X}$) $\to$ CHR($\mathcal{X}$) *le morphisme suivant, où* $G_u = \{t_1(\overrightarrow{v_1}), \ldots, t_m(\overrightarrow{v_m})\}$, $V_G = \{y_1, \ldots, y_p\}$ *et* $V(H, H') = \{x_1, \ldots, x_n\}$:

$$[\![H \backslash H' \Leftrightarrow G | B_{\mathcal{X}}, B_u]\!] = \begin{cases} H^*, H'^* \Rightarrow G_{\mathcal{X}} | \mathrm{id}\_i(K, x_1, \ldots, x_n), \\ \quad \mathrm{exists}(K, y_1), \ldots, \mathrm{exists}(K, y_p), \\ \quad \mathrm{ask}\_t_1(K, \overrightarrow{v_1}), \ldots, \mathrm{ask}\_t_m(K, \overrightarrow{v_m}). \\ H^* \backslash H'^*, \mathrm{id}\_i(K, x_1, \ldots, x_n), \\ \quad \mathrm{entailed}\_t_1(K, \overrightarrow{v_1}), \ldots, \mathrm{entailed}\_t_m(K, \overrightarrow{v_m}) \\ \quad \Rightarrow G_{\mathcal{X}} | B_{\mathcal{X}}, B^*_u. \end{cases}$$

Les contraintes utilisateur ask$_t_j$, exists, *tokentailed*$\_t_j$ et id$_i$ correspondent aux propositions atomiques introduites dans la sémantique déclarative. i est un identificateur unique associé à la règle. La sémantique en logique linéaire établit le lien entre la sémantique opérationnelle et la transformation définie ci-dessus. Cela conduit au résultat suivant qui prouve la correction de la transformation en sémantique logique linéaire :

Théorème 5 *Pour tout programme* CHRat($\mathcal{X}$) P *et requête* $T_0 \wedge c_0 \in \mathcal{L}'$:

$$\mathcal{LL}_P[T_0 \wedge c_0] = \mathcal{LL}_{[\![P]\!]}(T_0, C_0)$$

6 Une définition hiérarchique d'un solveur de contraintes sur les termes rationnels

6.1 Composant pour la contrainte union-find

L'algorithme classique du *union-find* (ou d'union d'ensembles disjoints) [19] a été implémenté en CHR [18] avec sa meilleure complexité algorithmique connue (temps quasi-linéaire). Atteindre cette complexité est un résultat notable pour un langage déclaratif, en particulier dans le domaine de la programmation logique [11]. L'algorithme *union-find* maintient une partition d'un univers, de sorte que chaque classe d'équivalence a un élément représentatif. Trois opérations agissent sur cette structure de données :

- `make(X)` ajoute l'élément `X` à l'univers, initialement au sein d'une classe réduite au singleton {`X`}.
- `find(X)` renvoie le représentant de la classe d'équivalence de `X`.
- `union(X,Y)` joint la classe d'équivalence de `X` avec celle de `Y` (en changeant éventuellement de représentant).

En tant que solveur de contraintes en logique classique, un tel algorithme résout la contrainte `A` $\simeq$ `B`. Les contraintes CHR `union` et `find` reflètent l'interprétation impérative de la structure de donnée du *union-find*, qui explicite les éléments représentatifs. Cependant, la propriété d'être un élément représentatif est une propriété non monotone qui ne peut pas être capturée par des contraintes en logique classique.

Implantation naïve en CHRat

Une première implantation repose sur une représentation classique des classes d'équivalence par des arbres enracinés [18]. Les racines sont les éléments représentatifs, elles sont marquées comme telles par la contrainte CHR `root(X)`. Les branches de l'arbre sont marquées par `A` $\rightsquigarrow$ `B`, où `A` est l'enfant et `B` le nœud parent.

```
component naive_union_find_solver.
export make/1, ≃/2.
make(A) <=> root(A).
union(A, B) <=>
    find(A, X), find(B, Y), link(X, Y).
A ⇝ B \ find(A, X) <=> find(B, X).
root(A) \ find(A, X) <=> X = A.
link(A, A) <=> true.
link(A, B), root(A), root(B) <=>
    B ⇝ A, root(A).
```

Cette implantation suppose que les points d'entrée `make` and `union` soit utilisés avec des arguments constants seulement, et que le premier argument de `find` soit toujours constant.

Poser la contrainte `A ≃ B` dans le store conduit à l'union des deux classes d'équivalence :

```
A ≃ B ==> union(A, B).
```

Une solution naïve d'implanter le solveur d'implication de ≃ consiste à suivre les branches jusqu'à trouver éventuellement un ancêtre commun pour `A` et `B`.

```
ask(K, A ≃ A) <=> entailed(K, A ≃ A).
A ⇝ C \ ask(K, A ≃ B) <=> C ≃
B | entailed(K, A ≃ B).
B ⇝ C \ ask(K, A ≃ B) <=> A ≃
C | entailed(K, A ≃ B).
```

Le calcul requis pour vérifier l'implication de contraintes est effectué en utilisant les gardes `C ≃ B` et `A ≃ C` récursivement. Soit S l'ensemble des éléments clos de $\mathcal{X}$.

Définition 6 *Un store c* est une structure valide pour *union-find si* `≃/2` *et* `root/1` *forme une forêt d'éléments dans* S.

Proposition 1 *La propriété de validité de la structure d'*union-find *est préservée par toutes les règles du solveur.*

Proposition 2 *Pour tout store* CHR *c avec une structure d'*union-find *valide et pour tous les éléments* `A` *et* `B`*,* `A` *et* `B` *partagent la même classe d'équivalence si et seulement si, pour toute variable fraîche* `K`*,* `entailed(K, A ≃B)` $\in \mathcal{O}_a($`ask(K, A ≃B)` $\wedge c)$.

Implantation optimisée en CHRat

La seconde implantation proposée dans [18] effectue les deux optimisations de *compression de chemins* (*path-compression*) et d'*union par rangs* (*union-by-rank*) pour atteindre la complexité quasi-linéaire en $O(n\alpha(n))$.

```
component union_find.
export make/1, ≃/2.
make(A) <=> root(A, 0).
union(A, B) <=>
    find(A, X), find(B, Y), link(X, Y).
A ⇝ B, find(A, X) <=> find(B, X), A ⇝
X.
root(A, _) \ find(A, X) <=> X = A.
link(A, A) <=> true.
link(A, B),root(A, N),root(B, M)<=>N≥M|
    B ⇝ A, N1 is max(M+1, N), root(A, N1).
link(B, A),root(A, N),root(B, M)<=>N≥M|
    B ⇝ A, N1 is max(M+1, N), root(A, N1).
```

L'implication optimisée pour la vérification de l'implication repose sur `find` pour trouver efficacement les représentants et ensuite les comparer. `check(K, A, B, X, Y)` représente la connaissance du fait que les représentants des classes d'équivalence de `A` et `B` sont les racines `X` et `Y` respectivement. Quand `X` et `Y` sont connus comme égaux, `entailed(K)` est posé dans le store :

```
ask(K, A ≃ B) <=>
    find(A, X), find(B, Y),
    check(K, A, B, X, Y).
root(X) \ check(K, A, B, X, X) <=>
    entailed(K).
```

Ces deux règles ne suffisent pas à définir un solveur complet pour l'implication parce que la structure d'arbres peut changer. En particulier, les racines trouvées pour `A` et `B` peuvent être invalidées par des appels ultérieurs à `union`, qui transformeront ces racines en sous-nœuds. Quand une ancienne racine devient un sous-nœud, les deux règles suivantes posent à nouveau `find` pour obtenir la nouvelle racine :

```
X ⇝ C \ check(K, A, B, X, Y) <=>
        find(A, Z), check(K, A, B, Z, Y).
Y ⇝ C \ check(K, A, B, X, Y) <=>
        find(B, Z), check(K, A, B, X, Z).
```

6.2 Composant pour la contrainte d'égalité entre arbres rationnels

Considérons maintenant les arbres rationnels, c'est-à-dire les arbres enracinés, ordonnés, non bornés, étiquetés et éventuellement infinis, avec un nombre fini de sous-arbres structurellement distincts [6]. Les nœuds sont supposés appartenir à l'univers considéré par le solveur *union-find*. Deux nœuds appartenant à la même classe d'équivalence sont supposés être structurellement égaux. Chaque nœud `X` a une signature `F/N` où `F` est l'étiquette de `X` et `N` est son arité : la contrainte associée est notée `fun(X, F, N)`. Pour chaque `I` entre 1 et `N`, la contrainte `arg(X, I, Y)` énonce que le `I`ème sous-arbre de `X` est (structurellement égal à) `Y`. Ces contraintes ont seulement à être compatible entre les éléments d'une même classe d'équivalence, ce que forcent les règles suivantes :

```
component rational_tree_solver.
import ≃/2 from union_find_solver.
export fun/3, arg/3, ∼/2.
fun(X0, F0, N0) \ fun(X1, F1, N1) <=>
    X0 ≃ X1 |
    F0 = F1, N0 = N1.
arg(X0, N, Y0) \ arg(X1, N, Y1) <=>
    X0 ≃ X1 |
    Y0 ≃ Y1.
```

La contrainte que deux arbres sont structurellement égaux, notée `X ∼ Y`, est réduite à l'union des deux classes d'équivalence :

```
X ∼ Y <=> X ≃ Y.
```

Le calcul associé à la vérification de l'implication de `A` $\sim$ `B` requiert une dérivation co-inductive des comparaisons structurelles pour casser les boucles. Ceci est fait par *mémoisation* : les jetons `checking(K, A, B)` signalent que `A` peut être supposé égal à `B` lorsque la vérification de cette égalité est déjà en cours. `checkTreeAux` vérifie que les signatures de `A` et de `B` sont égales et compare les arguments.

```
ask(K, A ∼ B) <=> checkTree(K, A, B).
checkTree(K, A, B) <=> eqTree(K, A, B) |
    entailed(K, A ∼ B).
ask(eqTree(K, A, B)) <=>
    checking(K, A, B),
    fun(A, FA, NA), fun(B, FB, NB),
    checkTreeAux(K, A, B, FA, NA, FB, NB).
checkTreeAux(K, A, B, F, N, F, N) <=>
    askArgs(K, A, B, 1, N),
    collectArgs(K, A, B, 1, N).
```

`askArgs` ajoute un jeton `askArg` par pair point-à-point de sous-arbres de `A` et de `B`. `askArg` répond `entailedArg` si tous correspondent. `collectArg` assure que tous les jetons `entailedArg` ont été posés avant de conclure sur l'implication de `eqTree(K, A, B)`.

```
askArgs(K, A, B, I, N) <=> I ≤ N |
    arg(A, I, AI), arg(B, I, BI),
    askArg(K, A, B, I, AI, BI),
    J is I + 1, askArgs(K, A, B, J, N).
askArgs(K, A, B, I, N) <=> true.
collectArgs(K, A, B, I, N),
    entailedArg(K, A, B, I) <=>
    J is I + 1,
    collectArgs(K, A, B, J, N).
collectArgs(K, A, B, I, N) <=> I > N |
    entailed(eqTree(K, A, B)).
```

`askArg` vérifie en premier lieu si l'égalité a été *mémoisée*, et sinon demande sa vérification :

```
checking(K, AI, BI) \
   askArg(K, A, B, I, AI, BI) <=>
   entailedArg(K, A, B, I).
askArg(K, A, B, I, AI, BI) <=>
   eqTree(K, AI, BI) |
   entailedArg(K, A, B, I).
```

Par simplicité, ce programme n'a pas de ramasse-miette. En particulier, les jetons mémoisés `checking(K,A,B)` ne sont jamais retirés du store, et les contraintes non impliquées ne sont pas éliminées.

7 Conclusion et perspectives

Nous avons montré qu'en laissant le programmeur définir en CHRat non seulement la vérification de satisfiabilité mais aussi la vérification d'implication de contraintes, cette version de CHR devient complètement modulaire, c'est-à-dire que les composants définis peuvent être réutilisés dans les règles et les gardes d'autres composants sans restriction. De plus, cette discipline de programmation n'est pas trop coûteuse pour le programmeur, le solveur peut se réduire à une simple introspection du store par `c \ ask(K, c) <=> entailed(K)`. Dans le cas général cependant, les règles CHRat pour `ask(K,c)` peuvent effectuer des calculs complexes arbitraires pour dériver le jeton de contraintes `entailed(K)`.

Les sémantiques opérationnelles et déclaratives de CHRat en logique linéaire ont été définies et prouvées équivalentes, et la transformation de programme de CHRat en CHR qui est à la base de notre compilateur a été prouvée correcte vis-à-vis de la sémantique en logique linéaire. Il est bon de noter que la transformation décrite est dirigée par la syntaxe (uniforme) et donc compatible avec d'autres préoccupations orthogonales concernant la modularité, comme les méthodologies pour faire collaborer plusieurs solveurs de contraintes [2]. Nous avons aussi montré que certains exemples classiques de solveurs de contraintes définis en CHR peuvent facilement être modularisés en CHRat et réutilisés pour construire des solveurs de contraintes complexes.

Comme perspectives futures, le cadre fourni par CHRat peut être amélioré de plusieurs façons. Les solveurs d'implication ne devraient jamais conduire à un échec et ne devraient pas interférer avec l'interprétation logique des contraintes exportées présentes dans le store CHR. L'exemple du *union-find* est un cas typique où le solveur d'implication change le store par compression de chemins tout en conservant la contrainte exportée $\simeq$ inchangée. La transformation d'un programme CHRat en un programme CHR classique fait que notre implantation bénéficie directement des optimisations des compilateurs CHR et peut suggérer de nouvelles optimisations. Alors que la gestion efficace des `ask` et `entailed` est laissée à l'implantation CHR sous-jacente, la gestion de la mémoire, du cache et de la mémoisation pour la vérification de l'implication est laissée au programmeur. De bonnes stratégies pour la récupération des miettes ainsi que pour la vérification de la non-implication restent à trouver, comme le montre le solveur pour les arbres rationnels. Enfin, le problème de compilation séparée n'a pas été discuté ici mais constitue un prolongement naturel.

Remerciements

Nous remercions Rishi Kumar pour son travail préliminaire dans cette voie avec le premier auteur, et Jacques Robin pour son récent intérêt pour ce travail.

Références

[1] Slim Abdennadher. Operational semantics and confluence of constraint propagation rules. In *Proceedings of CP'1997, 3rd International Conference on Principles and Practice of Constraint Programming*, volume 1330 of *Lecture Notes in Computer Science*, pages 252–266, Linz, 1997. Springer-Verlag.

[2] Slim Abdennadher and Thom W. Frühwirth. Integration and optimization of rule-based constraint solvers. In Maurice Bruynooghe, editor, *Logic Based Program Synthesis and Transformation. LOPSTR, Revised Selected Papers*, Lecture Notes in Computer Science. Springer, 2003.

[3] Slim Abdennadher and Heribert Schütz. CHRv : A flexible query language. In *FQAS '98 : Proceedings of the Third International Conference on Flexible Query Answering Systems*, pages 1–14, London, UK, 1998. Springer-Verlag.

[4] Hariolf Betz and Thom W. Frühwirth. A linear-logic semantics for constraint handling rules. In *Proceeding of CP 2005, 11th*, pages 137–151, 2005.

[5] Emmanuel Coquery and François Fages. A type system for CHR. In *Recent Advances in Constaints, revised selected papers from CSCLP'05*, number 3978 in Lecture Notes in Artificial Intelligence, pages 100–117. Springer-Verlag, 2006.

[6] B. Courcelle. Fundamental properties of infinite trees. *Theoretical Computer Science*, 25 :95–169, 1983.

[7] Gregory Duck, Maria Garcia de la Banda, and Peter Stuckey. Compiling ask constraints. In *Proceedings of International Conference on Logic Programming ICLP 2004*, Lecture Notes in Computer Science. Springer-Verlag, 2004.

[8] Gregory J. Duck, Peter J. Stuckey, Marìa Garcìa de la Banda, and Christian Holzbaur. Extending arbitrary solvers with constraint handling rules. In *Proceedings of PPDP'03, International Conference on Principles and Practice of Declarative Programming, Uppsala, Sweden*, pages 79–90. ACM Press, 2003.

[9] François Fages, Paul Ruet, and Sylvain Soliman. Linear concurrent constraint programming : operational and phase semantics. *Information and Computation*, 165(1) :14–41, February 2001.

[10] T. Frühwirth. Theory and practice of Constraint Handling Rules. *Journal of Logic Programming, Special Issue on Constraint Logic Programming*, 37(1-3) :95–138, October 1998.

[11] Harald Ganzinger. A new metacomplexity theorem for bottom-up logic programs. In *Proceedings of International Joint Conference on Automated Reasoning*, volume 2083 of *Lecture Notes in Computer Science*, pages 514–528. Springer-Verlag, 2001.

[12] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50(1), 1987.

[13] K.P. Girish and Sunil Jacob John. Relations and functions in multiset context. *Information Sciences*, 179(6) :758–768, 2009.

[14] Rémy Haemmerlé and François Fages. Modules for Prolog revisited. In *Proceedings of International Conference on Logic Programming ICLP 2006*, number 4079 in Lecture Notes in Computer Science, pages 41–55. Springer-Verlag, 2006.

[15] Pascal Van Hentenryck, Vijay A. Saraswat, and Yves Deville. Design, implementation, and evaluation of the constraint language cc(FD). *Journal of Logic Programming*, 37(1-3) :139–164, 1998.

[16] Vijay A. Saraswat. *Concurrent constraint programming*. ACM Doctoral Dissertation Awards. MIT Press, 1993.

[17] T. Schrijvers, B. Demoen, G. Duck, P. Stuckey, and T. Fruhwirth. Automatic implication checking for CHR constraint solvers. *Electronic Notes in Theoretical Computer Science*, 147 :93–111, January 2006.

[18] Tom Schrijvers and Thom W. Frühwirth. Analysing the CHR implementation of unionfind. In *Proceedings of the 19th Workshop on (Constraint) Logic Programming, WCLP'05*, Ulm, Germany, 2005.

[19] Robert Endre Tarjan and Jan van Leeuwen. Worst-case analysis of set union algorithms. *Journal of ACM*, 31(2) :245–281, April 1984.

Une approche basée sur la décomposition arborescente pour la résolution d'instances SAT structurées

Djamal Habet **Lionel Paris** **Cyril Terrioux**

LSIS - UMR CNRS 6168
Université Paul Cézanne (Aix-Marseille 3)
Avenue Escadrille Normandie-Niemen
13397 Marseille Cedex 20 (France)
{djamal.habet, lionel.paris, cyril.terrioux}@univ-cezanne.fr

Résumé

Le principal objectif de ce papier est de proposer une nouvelle approche pour la résolution d'instances SAT structurées. La structure d'une instance SAT peut être capturée par un hypergraphe dont les sommets correspondent à ses variables et les hyperarêtes à ses clauses. La méthode proposée est basée sur une décomposition arborescente de cet hypergraphe qui servira principalement à guider le processus d'énumération d'un algorithme de type DPLL. Durant la recherche, cette méthode détecte certaines informations et les enregistre sous forme de *goods* et *nogoods* structurels. Par l'exploitation de telles informations, cette méthode évite de visiter inutilement plusieurs fois une zone de l'espace de recherche préalablement explorée. Elle garantit ainsi une borne de complexité théorique en temps qui dépend de la décomposition arborescente utilisée. Dans la partie expérimentale, cette approche sera évaluée sur des instances SAT structurées et comparée à d'autres approches.

1 Introduction

Le problème de satisfiabilité (SAT) consiste à tester si une formule booléenne écrite sous la forme normale conjonctive est satisfiable. SAT est l'un des problèmes NP-complets les plus étudiés à cause de son importance aussi bien sur le plan théorique que pratique. Encouragées par les progrès impressionnants dans la résolution pratique du problème SAT, diverses applications allant de la vérification formelle à la planification sont encodées et résolues dans le formalisme SAT. Par ailleurs, la majorité des solveurs actuels et efficaces pour SAT sont basés sur des algorithmes de recherche de type *backtrack* construits autour de la procédure Davis-Putnam-Logemann-Loveland (DPLL) [5]. Ces algorithmes sont améliorés par plusieurs techniques permettant notamment d'élaguer l'espace de recherche. Parmi ces techniques, on peut citer l'apprentissage, l'utilisation de méthodes liées à la propagation des contraintes étendues, à l'exploitation des symétries, ... L'impact de ces différentes techniques dépend particulièrement de la nature et du type des instances considérées. Par exemple, l'apprentissage peut s'avérer très utile et pertinent lors de la résolution d'instances SAT structurées et peu efficace dans la résolution d'instances générées aléatoirement. Cependant, dans le pire des cas, la complexité en temps de la résolution de SAT par un algorithme basé sur DPLL est donnée par $O(m2^n)$, où n et m sont respectivement le nombre de variables et le nombre de clauses de l'instance traitée.

Améliorer l'efficacité des solveurs par l'exploitation de la structure du problème considéré a été largement étudiée en CSP (Constraint Satisfaction Problem) (par exemple dans [10]) et d'une manière bien moins prononcée en SAT. La structure d'un problème est alors représentée par un (hyper)graphe. Par exploitation de la structure d'un problème, nous entendons le fait de tirer profit des propriétés structurelles qui peuvent être représentées et capturées par les propriétés de cet (hyper)graphe. Concernant SAT, la structure de l'instance modélisant un problème (une application) du monde réel est le résultat de la conception modulaire de celui-ci, tel que ces modules possèdent entre eux une inter-

connexion minimale exprimée par un nombre minimal de variables [12]. Par conséquent, la décomposition du problème initial en un ensemble de sous-problèmes pourrait faciliter sa résolution.

Partant de ce constat, le but de notre travail est de fournir une nouvelle approche basée sur la décomposition arborescente de l'hypergraphe qui représente une instance SAT. Cette décomposition offre un ordre sur les variables du problème qui sera exploité par une méthode de résolution de type DPLL. De plus, durant la recherche, plusieurs informations qui correspondent soit à des échecs, soit à des succès lors des tentatives de résolution de(s) (sous-)problème(s) sont apprises sous forme de *goods* et *nogoods*. L'avantage attendu d'un tel apprentissage est de pouvoir élaguer l'espace de recherche et de ne pas résoudre plusieurs fois le même sous-problème. Une telle démarche nous permet également de fournir des bornes de complexité en temps et en espace pour la méthode proposée. Nous appuyons notre travail par la présentation de résultats expérimentaux et une comparaison avec certaines méthodes existantes de la littérature.

La papier est organisé comme suit. La section 2 introduit les notions de base et les notations nécessaires au reste du papier. Elle inclut notamment la définition formelle du problème SAT, de certains concepts liés à la théorie des graphe et un court rappel de l'algorithme DPLL. La section 3 présente notre approche basée sur la notion de décomposition arborescente, que nous appelons DPLL-TD, en posant d'abord les bases théoriques qui sont obligatoires pour ensuite décrire DPLL-TD et justifier sa validité. Nous fournirons également, dans cette même section, les complexités en temps et en espace de notre algorithme. La section 4 présente quelques détails sur l'implémentation de DPLL-TD et expose les résultats obtenus sur un certain nombre d'instances SAT structurées issues des précédentes compétitions SAT. La section 5 recense les travaux existants autour de la notion de décomposition arborescente dans le cadre principalement du problème SAT. Enfin, la section 6 discute et conclut ce travail.

2 Notations

Cette section est dédiée à la définition du problème SAT et des notations qui lui correspondent, à un rappel de l'algorithme DPLL ainsi qu'à l'introduction d'un certain nombre de concepts de la théorie des graphes nécessaires à la compréhension du papier.

2.1 Le problème de satisfiabilité

Une instance $\mathcal{F}$ du problème de satisfiabilité (SAT) est définie par le couple $\mathcal{F} = (\mathcal{X}, \mathcal{C})$, tel que $\mathcal{X}$ est un ensemble de variables booléennes (leurs valeurs appartiennent à l'ensemble $\{vrai,\ faux\}$) et $\mathcal{C}$ est un ensemble de clauses. Une clause est une disjonction finie de littéraux et un littéral est soit une variable, soit sa négation.

Pour un littéral donné l, $var(l) = \{v | l = v\ ou\ l = \neg v\}$ correspond à l'ensemble singleton de la variable sur laquelle porte l. Par ailleurs, un littéral peut-être considéré comme une clause contenant uniquement ce littéral, ce qui correspond à la définition de la clause unitaire. De plus, pour une clause donnée c, $var(c) = \cup_{l\ dans\ c} var(l)$ est l'ensemble des variables apparaissant dans c (positivement ou négativement). Par exemple, si $c = x_1 \vee \neg x_2 \vee \neg x_3$ alors nous avons $var(\neg x_2) = \{x_2\}$ et $var(c) = \{x_1,\ x_2,\ x_3\}$.

Une interprétation I des variables de $\mathcal{F}$ est définie par un ensemble de littéraux, tel que $\forall (l_1,\ l_2) \in I^2,\ l_1 \neq l_2,\ var(l_1) \neq var(l_2)$. Également, une variable qui apparaît positivement (respectivement négativement) dans I signifie qu'elle est fixée à $vrai$ (respectivement à $faux$). Aussi, l'interprétation I est dite partielle si $|I| < |\mathcal{X}|$, complète sinon (toutes les variables sont fixées). De plus, pour une interprétation donnée I d'un ensemble de variables $Y \subseteq \mathcal{X}$ et pour un sous-ensemble $Z \subseteq Y$, $I[Z]$ est la projection de I réduite aux variables de Z. Enfin, un modèle de $\mathcal{F}$ est une interprétation complète qui satisfait toutes les clauses de $\mathcal{F}$ et $Sol(\mathcal{F})$ désigne l'ensemble de tous les modèles de $\mathcal{F}$.

Par conséquence, le problème de satisfiabilité (SAT) consiste à tester si $\mathcal{F}$ possède un modèle ($Sol(\mathcal{F}) \neq \emptyset$). Si c'est le cas alors $\mathcal{F}$ est dite satisfiable, sinon $\mathcal{F}$ est insatisfiable.

2.2 L'algorithme DPLL

Introduite en 1960 et en dépit de sa simplicité, la procédure de Davis-Putnam-Logemann-Loveland (DPLL) [5] subsiste comme l'une des meilleures méthodes complètes pour SAT. La procédure DPLL est un algorithme de type *backtrack*. Elle construit un arbre de recherche binaire dans lequel chaque nœud est associé à un appel récursif. Toutes les feuilles, exceptées éventuellement une pour les instances satisfiables, représentent l'aboutissement de la procédure à une contradiction (correspondant à une clause vide notée par $\square$).

Plus précisément, à chaque itération (appel récursif), DPLL choisit une variable non interprétée (libre), lui affecte la valeur $vrai$ et simplifie $\mathcal{C}$. Si l'instance résultant de l'appel courant est satisfiable alors $\mathcal{F}$ est également satisfiable. Dans le cas contraire, DPLL affecte la valeur $faux$ à cette variable et le même processus est répété. L'opération de simplification consiste essentiellement à supprimer les clauses qui deviennent satisfaites sous l'interprétation courante I et à réduire les clauses qui contiennent des littéraux falsifiés (évalués à $faux$ sous I également).

Le choix de la variable de branchement est un facteur primordial de la procédure DPLL dont les performances dépendent directement. La littérature regorge de plusieurs

Algorithme 1 : DPLL(in : $\mathcal{C}$, I)

```
1  Propagation-unitaire (C, I)
2  si □ ∈ C alors retourner faux
3  sinon
4      si C = ∅ alors
5          retourner vrai
6      sinon
7          Choisir une variable libre v
8          si DP (C ∪ {v}, I ∪ {v}) alors retourner vrai
9          sinon
10             retourner DP (C ∪ {¬v}, I ∪ {¬v})
```

Algorithme 2 : Propagation-Unitaire(in/out : $\mathcal{C}$, I)

```
1  tant que Il n'existe pas de clauses vides et qu'il existe une clause unitaire l
   dans F faire
2      I ← I ∪ l (satisfaire l)
3      simplifier C
```

heuristiques de branchement qui sont généralement basées sur la propagation unitaire et le nombre d'occurrences des variables (sous leurs formes positives ou négatives) dans les clauses.

2.3 Théorie des graphes

Comme nous l'avons annoncé dans l'introduction, notre but est d'exploiter les propriétés structurelles d'une instance SAT exprimées par sa représentation sous forme d'un graphe, qui sera donc décomposé en conséquence. Nous allons formuler et rappeler ici quelques définitions liées à ces points.

Définition 1 *Soit l'instance SAT $\mathcal{F} = (\mathcal{X}, \mathcal{C})$. L'hypergraphe $H = (\mathcal{V}, \mathcal{E})$ qui caractérise l'instance $\mathcal{F}$ est défini tel que toute variable de $\mathcal{X}$ est représentée par un sommet de $\mathcal{V}$ et chaque clause $c \in \mathcal{C}$ est représentée par une hyper-arête $e \in \mathcal{E}$ (c'est-à-dire un sous-ensemble de $\mathcal{V}$) telle que $var(c) = e$.*

Par exemple, considérons l'instance SAT $\mathcal{F} = (\{x_1, x_2, x_3\}, \{x_1 \vee \neg x_2 \vee x_3,\ x_1 \vee \neg x_3,\ x_2 \vee x_3\})$. Selon la définition précédente, l'hypergraphe lui correspondant est $H = (\{x_1, x_2, x_3\}, \{\{x_1, x_2, x_3\},\ \{x_1, x_3\},\ \{x_2, x_3\}\})$.

Nous introduisons, à présent, la notion de graphe primal qui à un hypergraphe permet d'associer un graphe. Cette notion est nécessaire pour pouvoir exploiter, par la suite, la notion de décomposition arborescente qui est définie sur les graphes, et non sur les hypergraphes.

Définition 2 *Le graphe primal d'un hypergraphe $H = (\mathcal{V}, \mathcal{E})$ est le graphe $G_H = (\mathcal{V}, \mathcal{E}_H)$ tel que $\mathcal{E}_H = \{\{x, y\} | x, y \in V$ et $\exists e \in \mathcal{E}, \{x, y\} \subseteq e\}$.*

Reprenons la formule SAT $\mathcal{F}$ ci-dessous et sa représentation sous forme de l'hypergraphe H. Le graphe primal associé à H est donc $G_H = (\{x_1, x_2, x_3\}, \{\{x_1, x_2\},\ \{x_1, x_3\},\ \{x_2, x_3\}\})$.

On note par $G_{(\mathcal{X},\mathcal{C})}$ le graphe primal associé à une instance $(\mathcal{X}, \mathcal{C})$. C'est ce graphe qui sera finalement utilisé pour représenter la structure d'une instance SAT et pouvoir exploiter ses propriétés structurelles notamment au travers de la notion de décomposition arborescente définie ci-dessous [14].

Définition 3 *Soit $G_{(\mathcal{X},\mathcal{C})} = (\mathcal{V}, \mathcal{E}_H)$ le graphe primal associé à une instance $(\mathcal{X}, \mathcal{C})$. Une décomposition arborescente de $G_{(\mathcal{X},\mathcal{C})}$ est définie par le couple $(E, \mathcal{T})$, où d'une part $\mathcal{T} = (J, F)$ est un arbre dont l'ensemble des nœuds est représenté par J et l'ensemble des arêtes par F et d'autre part, $E = \{E_i : i \in J\}$ est une famille de sous-ensembles de $\mathcal{X}$, tel que chaque sous-ensemble (appelé cluster) E_i est un nœud de $\mathcal{T}$ et qui vérifie :*

(i) $\cup_{i \in J} E_i = \mathcal{V}$,

(ii) pour toute arête $\{x, y\} \in \mathcal{E}_H$, il existe $i \in J$ tel que $\{x, y\} \subseteq E_i$, et

(iii) pour tout $i, j, k \in J$, si k est sur une chaîne entre i et j dans $\mathcal{T}$, alors $E_i \cap E_j \subseteq E_k$.

La largeur de la décomposition arborescente $(E, \mathcal{T})$ est égale à $max_{i \in J}|E_i| - 1$. La largeur d'arbre (ou tree-width) w de $G_{(\mathcal{X},\mathcal{C})}$ est la plus petite largeur sur toutes les décompositions arborescente de $G_{(\mathcal{X},\mathcal{C})}$.

On note par $Desc(E_j)$ l'ensemble des variables appartenant à E_j ou à un descendant E_k de E_j. Aussi, soient E_i un cluster et E_j l'un de ses clusters fils, on désigne par $E_{par(j)}$ le cluster parent E_i de E_j et on suppose que $E_{par(1)} = \emptyset$ (E_1 étant le cluster racine). De plus, $\mathcal{C}[E_i]$ est l'ensemble des clauses appartenant exclusivement au cluster E_i. En d'autres mots, nous avons $\mathcal{C}[E_i] = \{c \in \mathcal{C} | var(c) \subseteq E_i$ et $var(c) \not\subseteq E_{par(i)}\}$.

Nous illustrons ces différentes définitions et notations à l'aide de l'exemple ci-dessous où la figure 1 correspond à une décomposition arborescente d'une une instance SAT donnée (non présentée ici). $E = \{E_i | i = 1, \ldots, 7\}$ est l'ensemble des clusters et E_1 est le cluster racine de cette décomposition arborescente. E_1 possède trois clusters fils E_2, E_3 et E_4, $E_2 = \{x_1, x_2, x_6, x_7\}$, $E_1 \cap E_2 = \{x_1,\ x_2\}$ (cette intersection est appelée *le séparateur* entre le cluster E_2 et son parent E_1), $E_{par(3)} = E_1$ et $Desc(E_3) = \{x_3, x_4, x_8, x_{15}, x_{18}, x_{20}\}$.

Avant de discuter les détails théoriques et algorithmiques de notre approche, nous allons d'abord donner ici l'idée de base qui est derrière. La décomposition arborescente de l'instance initiale divise celle-ci en parties (plus petites) et indépendantes, qui correspondent au clusters, mais qui restent toutefois interconnectées par les séparateurs. Résoudre

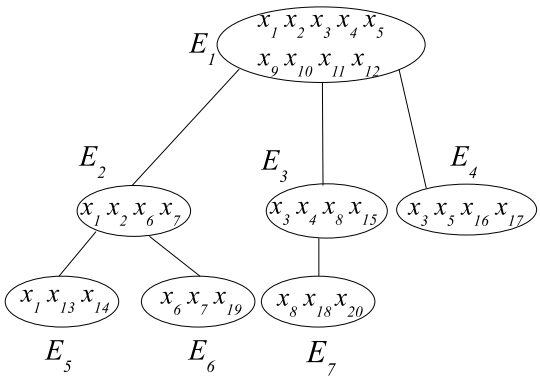


FIGURE 1 – Un exemple de décomposition arborescente.

l'instance initiale revient donc à résoudre ses différentes parties. Partant de la racine et par un parcours de type en profondeur d'abord, chaque cluster E_i est attaqué séparément. Si toutes les clauses restreintes à ce cluster sont satisfaites alors nous allons attaquer l'un de ses fils E_j (s'il existe) en tentant d'étendre l'interprétation des variables de E_i sur celles de E_j. Pour être plus précis, il est nécessaire de satisfaire les clauses de E_j en respectant les contraintes exprimées par l'interprétation des variables du séparateur entre E_i et E_j. En effet, ces variables sont partagées entre les deux clusters et par conséquence doivent être interprétées d'une manière identique. Si ce processus d'extension ne peut se faire alors on peut déduire qu'il ne sera jamais possible d'étendre l'interprétation des variables du séparateurs à un modèle. Plus intéressant encore, sauvegarder cette information peut-être utile pour éviter la répétition d'un même traitement dans le cas d'un retour-arrière au cluster E_i par exemple et l'occurrence de la même interprétation sur les variables du séparateur. De la même manière et pour les mêmes raisons, si les clauses du cluster E_i et celles de tous ses descendants sont satisfaites alors il est aussi utile de sauvegarder l'interprétation des variables du séparateur entre E_i et et son parent comme étant extensible à un modèle pour cette partie de l'instance. Pour résumé, nous appliquons une recherche de type *backtrack* sur la décomposition arborescente de l'instance SAT, tout en enregistrant toute information utile qui nous permettra d'élaguer l'espace de recherche.

3 Décomposition arborescente pour résoudre SAT

Cette section est la contribution majeure de notre travail, Dans un premier temps, elle souligne et explique les aspects théoriques liés à l'usage de la décomposition arborescente dans le cadre de SAT. Dans un second temps, elle introduit et détaille DPLL-TD qui est la traduction algorithmique de ces aspects théoriques. Enfin, nous exhibons des preuves sur les caractéristiques de DPLL-TD, à savoir qu'il est complet, correct et termine ainsi que sa complexité en temps et en espace.

3.1 Fondements théoriques

Dans la suite, nous considérons une instance SAT $\mathcal{F} = (\mathcal{X}, \mathcal{C})$ et une décomposition arborescente $(E, \mathcal{T})$ associée au graphe $G_{(\mathcal{X},\mathcal{C})}$.

Le but de ce premier théorème est de montrer que la décomposition arborescente de $G_{(\mathcal{X},\mathcal{C})}$ ne change en rien l'instance de départ (aucune information n'est perdue) et que chaque clause appartient à un et un seul cluster.

Théorème 1 *Les ensembles $(\mathcal{C}[E_i])_i$ forment une partition de $\mathcal{C}$.*

Preuve : De toute évidence, nous avons $\bigcup_i \mathcal{C}[E_i] \subseteq \mathcal{C}$. Démontrons maintenant que $\mathcal{C} \subseteq \bigcup_i \mathcal{C}[E_i]$.

Considérons une clause c. Par définition du graphe $G_{(\mathcal{X},\mathcal{C})}$, les sommets correspondant aux variables de $var(c)$ forment une clique de $G_{(\mathcal{X},\mathcal{C})}$. Ainsi, par définition de la décomposition arborescente, il existe au moins un cluster E_i tel que $var(c) \subseteq E_i$. En particulier, nous avons nécessairement un cluster E_k tel que $var(c) \subseteq E_k$ et $var(c) \not\subseteq E_{par(k)}$. Ainsi $c \in \mathcal{C}[E_k]$. D'où, $\bigcup_i \mathcal{C}[E_i] = \mathcal{C}$.

Maintenant, nous allons démontrer que pour des clusters quelconques E_i et E_j ($i \neq j$), $\mathcal{C}[E_i] \cap \mathcal{C}[E_j] = \emptyset$. Supposons que l'intersection n'est pas vide. Donc, il existe une clause c telle que $c \in \mathcal{C}[E_i]$ et $c \in \mathcal{C}[E_j]$. Il s'ensuit alors que $var(c) \subseteq E_i \cap E_j$. Par ailleurs, par définition de la décomposition arborescente, il existe une chaîne reliant E_i et E_j telle que chaque cluster rencontré sur cette chaîne contient $E_i \cap E_j$ et par conséquent $var(c)$. Sur une telle chaîne, on traversera nécessairement le père de E_i ou celui de E_j. Ainsi, nous avons $var(c) \subseteq E_{par(i)}$ ou $var(c) \subseteq E_{par(j)}$. Par conséquent, nous avons une contradiction puisque $c \in \mathcal{C}[E_i]$ et $c \in \mathcal{C}[E_j]$. Il en résulte que $\mathcal{C}[E_i] \cap \mathcal{C}[E_j] = \emptyset$.

En conclusion, les ensembles $(\mathcal{C}[E_i])_i$ forment effectivement une partition de $\mathcal{C}$. □

A partir des ensembles $\mathcal{C}[E_i]$, nous pouvons définir la notion de sous-problème induit :

Définition 4 *Soit E_i un cluster. Le sous-problème $\mathcal{F}_{E_i,I}$ enraciné en E_i et induit par l'interprétation I sur un sous-ensemble de $E_i \cap E_{par(i)}$ est défini par $(Desc(E_i), \bigcup_{E_k \subseteq Desc(E_i)} \mathcal{C}[E_k] \cup \{l | l \in I \text{ et } var(l) \subseteq E_i \cap E_{par(i)}\})$.*

En d'autres termes, $\mathcal{F}_{E_i,I}$ est constitué par toutes les clauses et variables du cluster E_i et celles des clusters de sa

descendance, avec des contraintes additionnelles exprimant l'interprétation courante des variable dans le séparateur entre E_i et son cluster parent. Ces contraintes sont formulées simplement par $\{l | l \in I \; et \; var(l) \subseteq E_i \cap E_{par(i)}\})$. Cette définition signifie aussi que résoudre le problème correspondant au sous-arbre enraciné à E_i doit respecter l'interprétation des variables qu'il partage avec son parent.

Propriété 1 *Considérons deux interprétations I et I' telles que $I \subseteq I'$ et un cluster E_i, nous avons alors $Sol(\mathcal{F}_{E_i,I'}) \subseteq Sol(\mathcal{F}_{E_i,I})$.*

Preuve : La seule différence entre les sous-problèmes $\mathcal{F}_{E_i,I'}$ et $\mathcal{F}_{E_i,I}$ est que le premier possède un certain nombre de clauses supplémentaires, notamment les clauses unitaires v tel que $v \in I' - I$ et $var(v) \subseteq E_i \cap E_{par(i)}$. Ainsi, nous avons nécessairement $Sol(\mathcal{F}_{E_i,I'}) \subseteq Sol(\mathcal{F}_{E_i,I})$. □

Cette propriété permet donc de caractériser la relation existant entre les ensembles de modèles de deux sous-problèmes enracinés en un même cluster E_i mais induit par deux interprétations différentes dont l'une est une extension de l'autre.

La définition suivante présente maintenant la notion de variable d'indépendance :

Définition 5 *Une variable v est une variable d'indépendance s'il existe un cluster E_i et deux de ses fils E_j et E_k tels que $v \in E_i \cap E_j \cap E_k$. On désigne par $\mathcal{X}_{ind}$ l'ensemble des variables d'indépendance de $\mathcal{F}$ par rapport à la décomposition arborescente considérée.*

Les variables d'indépendance jouent clairement un rôle particulier dans la décomposition arborescente. Plus précisément, pour pouvoir décomposer de façon valide un problème SAT en différents sous-problèmes indépendants, nous devons garantir que ces variables sont interprétées avec la même valeur dans les différents sous-problèmes dans lesquels elles interviennent. Par exemple, la décomposition arborescente de la figure 1 possède une seule variable d'indépendance x_3 localisée dans l'intersection de E_1 avec deux de ses fils E_3 et E_4. Donc, $\mathcal{X}_{ind} = \{x_3\}$. Si x_3 n'est pas interprétée lors de traitement de E_1 (rappelons qu'il est possible de satisfaire un ensemble de clauses sans nécessairement fixer toutes les variables apparaissant dans celles-ci) alors x_3 doit-être interprétée avec la même valeur lors du traitement des problèmes enracinés en E_3 et E_4. Le théorème suivant formalise l'indépendance entre sous-problèmes en termes de variables d'indépendance.

Théorème 2 *Soient un cluster E_i, E_j et E_k deux fils de E_i et une interprétation I d'un sous-ensemble Y de E_i. Si $\mathcal{X}_{ind} \cap E_i \subseteq Y$, alors les sous-problèmes $\mathcal{F}_{E_j,I[E_i \cap E_j]}$ et $\mathcal{F}_{E_k,I[E_i \cap E_k]}$ sont indépendants.*

Preuve : A partir de la définition de la décomposition arborescente, nous avons $Desc(E_j) \cap Desc(E_k) = E_j \cap E_k \subseteq E_i$. Clairement, nous avons $E_j \cap E_k \subseteq \mathcal{X}_{ind}$ et par conséquent $E_j \cap E_k \subseteq \mathcal{X}_{ind} \cap E_i$. Puisque les variables de $\mathcal{X}_{ind} \cap E_i$ sont interprétées dans I, cela assure que ces mêmes variables ont les mêmes interprétations sur n'importe quelle interprétation qui satisfait $\mathcal{F}_{E_j,I[E_i \cap E_j]}$ et $\mathcal{F}_{E_k,I[E_i \cap E_k]}$, au regard des clauses unitaires v telles que $v \in I$ et $var(v) \subseteq E_i \cap E_j \cap E_k$. De plus, ces clauses unitaires sont les seules à appartenir à la fois à $\mathcal{F}_{E_j,I[E_i \cap E_j]}$ et à $\mathcal{F}_{E_k,I[E_i \cap E_k]}$ (par application du théorème 1). Ainsi, les problèmes $\mathcal{F}_{E_j,I[E_i \cap E_j]}$ et $\mathcal{F}_{E_k,I[E_i \cap E_k]}$ sont bien indépendants. □

Le corollaire suivant établit que la satisfiabilité de sous-problèmes frères conduit à la satisfiabilité du sous-problème enarciné en leur cluster parent.

Corollaire 1 *Soient un cluster E_i et une interprétation I sur un sous-ensemble Y de E_i, si $\mathcal{X}_{ind} \cap E_i \subseteq Y$, I satisfait les clauses de $\mathcal{C}[E_i]$, et pour chaque fils E_j de E_i, le sous-problème $\mathcal{F}_{E_j,I[E_i \cap E_j]}$ est satisfiable, alors I peut-être étendue à un modèle de $\mathcal{F}_{E_i,I[E_i \cap E_{par(i)}]}$.*

Preuve : Soit M_{E_j} un modèle pour le sous-problème $\mathcal{F}_{E_j,I[E_i \cap E_j]}$. Par définition de $\mathcal{F}_{E_j,I[E_i \cap E_j]}$, il n'existe pas de variables v tel que $I \cup M_{E_j}$ contienne deux littéraux opposés à v. Selon le théorème 2, pour tout fils E_j et E_k de E_i, $\mathcal{F}_{E_j,I[E_i \cap E_j]}$ et $\mathcal{F}_{E_k,I[E_i \cap E_k]}$ sont indépendants. Ainsi, l'interprétation $I \cup \bigcup_{E_j \in fils(E_i)} M_{E_j}$ satisfait à la fois les clauses de $\mathcal{C}[E_i]$ et celles de $\mathcal{F}_{E_j,I[E_i \cap E_j]}$ pour chaque fils E_j de E_i. De cette manière, I peut-être étendue à un modèle de $\mathcal{F}_{E_i,I[E_i \cap E_{par(i)}]}$.□

Comme nous l'avons expliqué juste avant le début de cette section, nous voulons également garder trace de toute information utile qui nous permettra d'éviter plusieurs fois un même traitement, principalement la résolution récurrente de sous-problèmes. Cela revient à effectuer des coupes sur les branches de l'arbre explorant l'espace de recherche en présence d'informations indiquant si la satisfiabilité d'un sous-problème donné est déjà connue ou pas. De telles informations correspondent aux *goods* et *nogoods* que nous définissons comme suit :

Définition 6 *Etant donné un cluster E_i, une interprétation I sur un sous-ensemble de $E_i \cap E_{par(i)}$ est un* good *(respectivement un* nogood*) structurel de E_i si toute extension de I sur $E_i \cap E_{par(i)}$ peut-être étendue à un modèle de $\mathcal{F}_{E_i,I}$ (respectivement si $Sol(\mathcal{F}_{E_i,I}) = \emptyset$).*

En d'autres termes, un *good* (respectivement un *nogood*) structurel est une interprétation I d'un sous-ensemble de $E_i \cap E_{par(i)}$ qui peut (respectivement ne peut pas) être étendue à un modèle pour $\mathcal{F}_{I,E_i}$.

Propriété 2 *Soient un cluster E_i et un sous-ensemble $Y \subseteq X$ tel que $Desc(E_i) \cap Y \subseteq E_i \cap E_{par(i)}$. Pour tout* good *$g$ de E_i, toute interprétation I sur Y peut-être étendue à un modèle de $\mathcal{F}_{E_i, I[E_i \cap E_{par(i)}]}$ s'il existe une extension e_g de g sur $E_i \cap E_{par(i)}$ telle que $I[E_i \cap E_{par(i)}] \subseteq e_g$.*

Preuve : Par définition d'un *good*, e_g et par conséquent $I[E_i \cap E_{par(i)}]$ (puisque $I[E_i \cap E_{par(i)}] \subseteq e_g$) peuvent être étendues à un modèle M de $\mathcal{F}_{E_i,g}$. Il s'ensuit que M satisfait les clauses de $\bigcup_{E_k \subseteq Desc(E_i)} \mathcal{C}[E_k]$. De plus, il satisfait également les clauses unitaires de $\{v | v \in I \ et \ var(v) \subseteq E_i \cap E_{par(i)}\}$ puisque $I[E_i \cap E_{par(i)}] \subseteq e_g \subseteq M$. Ainsi, M est un modèle de $\mathcal{F}_{E_i, I[E_i \cap E_{par(i)}]}$ et $I[E_i \cap E_{par(i)}]$ peut-être étendue à un modèle de $\mathcal{F}_{E_i, I[E_i \cap E_{par(i)}]}$. □

La propriété précédente nous donne les conditions qui nous autorisent à effectuer une coupe selon les *goods* enregistrés. De la même manière, la propriété suivante exprime la condition de coupe par les *nogoods*.

Propriété 3 *Soient un cluster E_i et un sous-ensemble $Y \subseteq X$ tel que $Desc(E_i) \cap Y \subseteq E_i \cap E_{par(i)}$. Pour tout* nogood *$ng$ de E_i, aucune interprétation I sur Y tel que $ng \subseteq I$ ne peut-être étendue à un modèle de $\mathcal{F}_{E_i, I[E_i \cap E_{par(i)}]}$.*

Preuve : Nous avons $Sol(\mathcal{F}_{E_i, I[E_i \cap E_{par(i)}]}) \subseteq Sol(\mathcal{F}_{E_i, ng})$ (propriété 1). Par ailleurs, par définition d'un *nogood*, $Sol(\mathcal{F}_{E_i, ng}) = \emptyset$. Donc $Sol(\mathcal{F}_{E_i, I[E_i \cap E_{par(i)}]}) = \emptyset$ et I ne peut-être donc étendue à un modèle sur $Desc(E_i)$. □

Maintenant que nous avons apporté et démontré tous les éléments théoriques liés à notre approche, nous décrivons dans la section suivante leur exploitation algorithmique.

3.2 L'algorithme DPLL-TD

Considérons une instance SAT définie par $\mathcal{F} = (\mathcal{X}, \mathcal{C})$ et la décomposition arborescente $(E, \mathcal{T})$ du graphe primal obtenu à partir de la représentation en hypergraphe de $\mathcal{F}$. L'algorithme 3 décrit la méthode DPLL-TD (pour *DPLL with Tree Decomposition*) pour résoudre le problème de satisfiabilité. Durant la recherche, DPLL-TD tente d'étendre l'interprétation courante à un modèle, s'il existe. Pour cela, et en plus de l'ordre dicté par l'heuristique de choix de variables de branchement d'une procédure de type DPLL, la méthode DPLL-TD se sert de la décomposition arborescente $(E, \mathcal{T})$ afin d'affiner ce choix comme cela sera expliqué dans la suite. De plus, DPLL-TD exploite des informations préalablement apprises sous formes de *goods* (enregistrés dans $\mathcal{G}$) et de *nogoods* (enregistrés dans $\mathcal{N}$) dans le but d'élaguer l'espace de recherche.

Ainsi décrit, l'appel DPLL-TD($\mathcal{C}, I, E_i, \mathcal{G}, \mathcal{N}$) renvoie $vrai$ (respectivement $faux$) si le sous-problème $\mathcal{F}_{E_i, I}$ enraciné en E_i est satisfiable (respectivement insatisfiable).

Avant de détailler l'algorithme DPLL-TD, quelques notations et précisions supplémentaires sont nécessaires en plus de celles préalablement introduites. Les *goods* correspondant à un cluster donné E_i sont représentés et enregistrés dans l'ensemble $\mathcal{G}_{E_i}$ et $\mathcal{G}$ est un ensemble défini par $\mathcal{G} = \{\mathcal{G}_{E_i} | E_i \in E\}$. Par ailleurs, $\mathcal{N}$ désigne l'ensemble des *nogoods*.

Algorithme 3 : DPLL-TD(in : $\mathcal{C}$, I, E_i, in/out : $\mathcal{G}, \mathcal{N}$)

```
1  Propagation-unitaire (C ∪ N, I)
2  si □ ∈ C ∪ N alors retourner faux
3  sinon
4      si C ∪ N = ∅ alors retourner vrai
5      sinon
6          si (C ∪ N)[E_i] = ∅ et X_ind ∩ E_i = ∅ alors
7              sat ← vrai
8              S ← Fils(E_i)
9              tant que sat et S ≠ ∅ faire
10                 Choisir un cluster E_j dans S
11                 S ← S − {E_j}
12                 si ∃g ∈ G_{E_j}, ∃e_g extension de g sur
                   E_i ∩ E_j, I[E_i ∩ E_j] ⊆ e_g alors
13                     I ← I ∪ g
14                 sinon
15                     sat ← DPLL-TD (C, I, E_j, G, N)
16                     si sat alors
17                         Soit g ∈ G_{E_j} le dernier good enregistré
18                         I ← I ∪ g
                           sinon N ← N ∪ { ⋁_{l_k ∈ I[E_i ∩ E_j]} ¬l_k }
19
20             si sat alors G_{E_i} ← G_{E_i} ∪ {I[E_i ∩ E_par(i)]}
21             retourner sat
22         sinon
23             Choisir une variable non interprétée v dans E_i
24             si DPLL-TD (C ∪ {v}, I ∪ {v}, E_i, G, N) alors
25                 retourner True
26             sinon retourner DPLL-TD
               (C ∪ {¬v}, I ∪ {¬v}, E_i, G, N)
```

Le premier appel à DPLL-TD est fait avec les paramètres $(\mathcal{C}, \emptyset, E_1, \mathcal{G}, \mathcal{N})$, où E_1 est le cluster racine de la décomposition arborescente. En effet, au départ, aucune variable n'est interprétée. De même, aucun *(no)good* n'est connu (pour chaque cluster E_i, $\mathcal{G}_{E_i} = \emptyset$ et $\mathcal{N} = \emptyset$).

La première étape de cet algorithme (ligne 1) consiste à propager les clauses unitaire appartenant à $\mathcal{C} \cup \mathcal{N}$. En conséquence, si une clause vide est déduite alors l'algorithme retourne $faux$ signifiant que $\mathcal{F}$ est insatisfiable. Sinon, si $\mathcal{C} \cup \mathcal{N}$ est vidée alors $\mathcal{F}$ est satisfiable et l'algorithme retourne $true$ (ligne 4).

Maintenant, considérons l'ensemble des clauses $(\mathcal{C} \cup \mathcal{N})[E_i]$ et l'ensemble des variables d'indépendance restreintes au cluster courant E_i, $\mathcal{X}_{ind} \cap E_i$. Si l'un de ces deux ensembles n'est pas vide alors l'algorithme 3 procède à une énumération DPLL classique sur E_i dans le but d'interpréter ces variables (lignes 23 à 26). Par exemple, dans la figure 1, $x_3 \in \mathcal{X} \cap E_1$. Si x_3 n'est pas fixée alors une énumération sur x_3 (et éventuellement sur les variables non interprétées de $(\mathcal{C} \cup \mathcal{N})[E_1]$) est effectuée afin d'assurer la même interprétation de x_3 dans les clusters E_1, E_2 et E_3.

Ainsi, on garantit l'indépendance des sous-problèmes enracinés en E_3 et E_4 (corollaire 1).

Dans le cas contraire ($(\mathcal{C} \cup \mathcal{N})[E_i] = \emptyset$ et $\mathcal{X}_{ind} \cap E_i = \emptyset$), DPLL-TD traite les sous-problèmes enracinés en chaque fils E_j de E_i (pour rappel, si tous les sous-problèmes $\mathcal{F}_{E_j, I[E_i \cap E_j]}$ sont satisfaits alors $\mathcal{F}_{E_i, I}$ est également satisfait).

Lors du traitement des fils de E_i, les *goods* et *nogoods* déjà enregistrés peuvent être utiles. En effet, DPLL-TD vérifie s'il existe un *good* g dans $\mathcal{G}_{E_j}$ entre E_j (fils de E_i) et E_i qui respecte la condition de coupe donnée par la propriété 2 (ligne 12). Si un tel *good* g existe alors $\mathcal{F}_{E_j, I[E_i \cap E_j]}$ est satisfiable et il n'est pas nécessaire d'étudier à nouveau sa satisfiabilité. Inversement, si la satisfiabilité de $\mathcal{F}_{E_j, I[E_i \cap E_j]}$ est inconnue, elle sera déterminée par l'appel récursif DPLL-TD($\mathcal{C}, I, E_j, \mathcal{G}, \mathcal{N}$). Dans ce dernier cas, si $\mathcal{F}_{E_j, I[E_i \cap E_j]}$ est insatisfiable (le dernier appel renvoie $faux$), un *nogood* est détectée entre E_j et E_i. Celui-ci est représenté et enregistré comme une nouvelle clause définie par $\vee_{l_k \in I[E_i \cap E_j]} \neg l_k$ qui est ensuite rajoutée à l'ensemble des *nogoods* $\mathcal{N}$ (ligne 19). Revenons à la figure 1 et considérons le cluster E_1, son fils E_2 et une interprétation sur $E_1 \cap E_2 = \{x_1, x_2\}$ qui est $I[\{x_1, x_2\}] = \{\neg x_1, x_2\}$. Si $F_{2, \{\neg x_1, x_2\}}$ est insatisfiable alors un *nogood* est détecté et sauvegardé par la nouvelle clause $x_1 \vee \neg x_2$ dans $\mathcal{N}$.

Lors de l'occurrence de telles contradictions, DPLL-TD effectue un retour-arrière sur les variables de E_i et teste l'existence d'une nouvelle interprétation qui satisfait $(\mathcal{C} \cup \mathcal{N})[E_i]$. A noter que même si les clauses apprises (correspondant aux *nogoods*) sont sauvegardées dans un ensemble distinct $\mathcal{N}$, elles sont en pratique considérées comme une clause quelconque de $\mathcal{C}$. Nous effectuons cette distinction seulement dans le but de mettre en avant les *nogoods* appris tout le long de la recherche. Par ailleurs, la représentation sous forme clausale des *nogoods* nous dispense de l'application d'un traitement particulier lors de leur exploitation (incluant le test sur la possibilité de coupe dans l'arbre de recherche grâce aux *nogoods* appris). Ainsi, l'usage des *nogoods* est entièrement transparent dans DPLL-TD. Par ailleurs, ces clauses apprises permettent d'améliorer l'effet filtrant des propagations unitaires.

Finalement, les lignes 13 et 18 correspondent à l'extension de l'interprétation courante par le *good* utilisé par la coupe (par la propriété 2). Ainsi, I est étendue par l'ajout des littéraux du *good* g à ceux de I assurant un enregistrement d'un *good* valide sur E_i si $\mathcal{F}_{E_i, I}$ est satisfiable (ligne 20), où un *good* correspond à l'interprétation obtenue sur les variables de $E_i \cap E_{par(i)}$. A noter qu'aucune contradiction ne peut-être trouvée lors de cette extension, car les variables d'indépendance sont préalablement fixées comme expliqué précédemment. Par exemple, dans la figure 1, considérons le cluster E_2, son fils E_6 et une interprétation sur $E_2 \cap E_6 = \{x_6, x_7\}$ qui est $I[\{x_6, x_7\}] = \{x_6\}$. De plus, supposons qu'il existe un *good* $g = \{x_6, \neg x_7\} \in \mathcal{G}_{E_6}$. Par l'application de la propriété 2, $F_{E_6, \{x_6\}}$ est satisfiable et l'interprétation courante est étendue par $\{\neg x_7\}$.

3.3 Propriétés de DPLL-TD

Après avoir détailler le déroulement de l'algorithme DPLL-TD, nous allons nous intéresser à sa complexité en temps et en espace, sa complétude, sa validité et sa terminaison.

Théorème 3 *DPLL-TD est correct, complet et termine.*

Preuve : DPLL-TD diffère de DPLL par l'enregistrement des *goods* et des *nogoods* structurels utilisés afin d'éviter les redondances dans la recherche. Puisque DPLL est correct, complet et termine, il reste à prouver que les opérations supplémentaires effectuées par DPLL-TD ne remettent pas en cause ces propriétés.

Supposons que DPLL-TD($\mathcal{C}, I, E_i, \mathcal{G}, \mathcal{N}$) est l'appel courant. On veut vérifier si le sous-problème $\mathcal{F}_{E_i, I[E_i \cap E_{par(i)}]}$ est satisfiable. Pour cela, considérons le cas où I satisfait toutes les clauses de $(\mathcal{C} \cup \mathcal{N})[E_i]$. DPLL-TD essaiera d'étendre I sur chacun des fils de E_i. Soit E_j l'un de ces fils. Si la condition de la ligne 12 est vérifiée alors (et grâce à la propriété 2) $\mathcal{F}_{E_j, I[E_j \cap E_i]}$ est satisfiable. Dans le cas contraire, la satisfiabilité de $\mathcal{F}_{E_j, I[E_j \cap E_i]}$ est inconnue et pour la déterminer, il faut faire un nouvel appel à DPLL-TD par DPLL-TD($\mathcal{C}, I, E_j, \mathcal{G}, \mathcal{N}$). Si ce dernier renvoie $faux$, cela signifie que $\mathcal{F}_{E_j, I[E_j \cap E_i]}$ ne possède pas de modèle. De ce fait, l'opération d'enregistrement d'un nouveau *nogood* sous la forme d'une clause $\bigvee_{l_k \in I[E_i \cap E_j]} \neg l_k$ est valide. Etant donné qu'un *nogood* structurel n'est qu'un cas particulier d'un *nogood* classique, son usage comme n'importe quelle clause de C est aussi valide.

Par ailleurs, à la fin de la boucle **tant que**, si I peut-être étendue à un modèle sur tous ses fils, alors (et par l'application du corollaire 1), $I[E_i \cap E_{par(i)}]$ peut-être étendue à un modèle de $\mathcal{F}_{E_i, I[E_i \cap E_{par(i)}]}$. De plus, puisque I était étendue par les valeurs des *goods* de ses clusters fils, alors l'enregistrement du *good* $I[E_i \cap E_{par(i)}]$ pour le cluster E_i est aussi une opération valide.

Pour conclure, du fait que l'enregistrement et l'utilisation des *goods* et *nogoods* sont valides, DPLL-TD est correct, complet et termine. □

Théorème 4 *DPLL-TD a une complexité en $O((|E| + m).2^{w+s})$ en temps et en $O(|E|.s.2^s)$ en espace pour une instance SAT possédant n variables et m clauses et une décomposition arborescente $(E, \mathcal{T})$ de largeur w et dont la taille de la plus grande intersection entre deux clusters est s.*

Preuve : Considérons un cluster E_j et une interprétation partielle I sur $E_j \cap E_{par(j)}$ et essayons d'étendre I sur E_j en supposant (dans un soucis de simplicité) que l'ordre des variables est statique. Si $E_j = E_1$ (le cluster racine de la décomposition arborescente), $I = \emptyset$ et DPLL-TD calculera au plus $2^{|E_1|+1}$ interprétations partielles pour étendre I. Sinon, il existe au plus $2^{|E_j|-|I|+1}$ interprétations partielles pour étendre I. Puisque, il y a au maximum $2^{|I|}$ interprétations partielles possibles de taille $|I|$ et que le sous-problème $\mathcal{F}_{E_j,I}$ n'est résolu qu'une fois grâce aux *goods* et *nogoods* enregistrés, le nombre maximum total d'interprétations partielles étudiées par DPLL-TD pour un cluster E_j est $\sum_{|I|=0}^{s} 2^{|E_j|-|I|+1}.2^{|I|} = s.2^{|E_j|+1}$. De plus, DPLL-TD va évaluer au maximum $O(2^{w+2})$ interprétations car le cluster E_j est indépendant de son cluster père du fait que les variables d'indépendance dans $\mathcal{X}_{ind} \cap E_j \cap E_{par(j)}$ sont déjà interprétées. Par ailleurs, pour chacune des interprétations partielles, DPLL-TD applique la propagation des clauses unitaires en $O(m + |\mathcal{N}|)$. Pour un séparateur donné de taille k (un séparateur est défini par l'intersection d'un cluster avec son père), le nombre d'interprétations partielles est borné par 2^{k+1}. Donc, le nombre total de *(no)goods* est donné par $|E|.2^{s+1}$. Finalement, l'enregistrement d'un *(no)good* peut se faire en $O(s)$ et le test de la condition de la ligne 12 en $O(s.2^{s+1})$. Ainsi, la complexité en temps de DPLL-TD est $O((m + |E|.2^{s+1}).2^{w+1} + s.2^{s+1}.2^{w+1})$, autrement dit $O((m + |E|).2^{w+s})$.

Concernant la complexité en espace, celle-ci dépend uniquement des *goods* et des *nogoods* enregistrés. Comme l'espace nécessaire pour la sauvegarde d'un *(no)good* est $O(s)$, la complexité en espace de DPLL-TD est $O(|E|.s.2^s)$. □

En d'autres termes, la complexité de DPLL-TD dépend des caractéristiques de la décomposition arborescente, à savoir sa largeur et la taille du plus grand cluster. Comparé à un algorithme DPLL classique, la complexité de ce dernier dépend du nombre de variables de l'instance traitée.

4 Résultats préliminaires

Cette section présente quelques détails sur l'implémentation de DPLL-TD et montre des résultats préliminaires obtenus sur des instances SAT issues des précédentes compétitions SAT, de 2002 à 2007 (`www.satcompetition.org`). Nous comparons DPLL-TD à 4 solveurs performants : Minisat[6], Rsat[13], Zchaff [16] et Satz [11] et toutes les instances ont été prétraitées avec SatElite [7]. L'algorithme DPLL-TD est codé sur la base de Satz . Ce choix est fait par notre bonne connaissance de son code source que nous avons toutefois remanié afin de prendre en compte les spécificités de notre approche, comme par exemple l'enregistrement de *nogoods* sous forme de clauses. Satz est donc utilisé comme algorithme d'énumération sur les clusters et les variables d'indépendance. Concernant le calcul de la décomposition arborescente, il repose sur une triangulation heuristique du graphe primal. Cette triangulation est effectuée grâce à la méthode Min-fill [15] qui s'avère être une des méthodes de triangulation heuristiques les plus robustes. Les expérimentations sont faites sous Linux sur un PC avec un processeur Pentium 4 à 3,2 GHz et une mémoire vive de 1 Go. Chaque instance est exécutée durant 600 secondes au maximum pour chacun des solveurs. Nous avons testé près de 1800 instances et nous présentons dans la table 1 une sélection des résultats les plus pertinents. La première colonne est le nom du benchmark et la deuxième le nombre d'instances que nous avons testé pour ce benchmark. Le reste de la table donne, pour chaque solveur, le nombre d'instances (#r) qu'il a pu résoudre ainsi que le somme des temps (t) en secondes nécessaires à les résoudre.

Au regard de la dernière colonne, qui donne le classement de DPLL-TD par rapport aux autres solveurs en termes de nombre d'instances résolues, nous pouvons annoncer que notre solveur est compétitif et améliore même les performances de deux solveurs majeurs, Minisat et Rsat, sur quelques instances. Egalement et comparativement à Satz, DPLL-TD améliore significativement les résultats de ce dernier. Ce qui peut nous laisser croire que l'implémentation de notre méthode sur la base d'un solveur plus récent et plus performant nous permettra encore d'obtenir des résultats de meilleure qualité.

Nous avons observé que les instances les plus facilement résolues par DPLL-TD sont majoritairement insatisfiables. Aussi, une première explication possible pour ce phénomène est que DPLL-TD, grâce à la décomposition arborescente qu'il exploite, est capable d'aborder la recherche par un cluster ayant peu ou pas de modèles. Si cette explication est certainement la bonne pour des instances de petite taille, nous avons pu observer que, dans la plupart des cas, DPLL-TD produit, mémorise et exploite un nombre conséquent de *goods* et de *nogoods* structurels. Aussi, les bons résultats observés s'explique également par l'apport des *goods* et des *nogoods* structurels, et plus précisément par les redondances dans l'espace de recherche qu'ils permettent d'éviter.

A la vue de ces premières expérimentations, il est possible d'annoncer que les résultats sont très encourageants et de suggérer que le comportement de DPLL-TD peut-être encore perfectionné par la prise en compte de différents paramètres, comme un meilleur choix de décomposition arborescente, un choix plus éclairé de cluster racine, ...

5 Travaux connexes

La structure des problèmes a été utilisée pour définir des heuristiques d'ordre sur les variables dans des méthodes de type backtrack depuis que Freuder [8] a présenté une fa-

		DPLL-TD		Minisat		Rsat		Zchaff		Satz		Classement de
Benchs.	# inst.	#r	t	#r	t	#r	t	#r	t	#r	t	DPLL-TD
linvrinv	8	3	1	3	1	3	2	3	9	3	1	1
mod2-3cage-unsat	23	6	2370	**18**	2579	1	540	0	-	0	-	2
mod2-rand3bip-sat	33	1	351	**17**	2534	6	1112	7	1646	8	1923	5
mod2-rand3bip-unsat	15	**11**	793	9	1062	6	872	6	1124	0	-	1
mod2c-3cage-unsat	6	**4**	115	3	563	0	-	0	-	0	-	1
mod2c-rand3bip-sat	33	0	-	17	2559	11	1410	**18**	3156	0	-	5
mod2c-rand3bip-unsat	15	**15**	676	10	1810	8	1337	6	1125	0	-	1
clqcolor	16	7	181	6	171	**9**	36	**9**	23	0	-	3
fclqcolor	16	9	136	9	169	9	22	9	11	0	-	1
fphp	42	16	617	**17**	165	16	49	16	28	**17**	555	3
php	42	16	123	6	258	16	134	12	231	**18**	721	2
sorge05	104	36	1752	**69**	5571	59	4347	54	2748	33	1316	4
driverLog	36	32	1	**36**	< 1	**36**	< 1	**36**	< 1	**36**	2	5
Ferry	36	20	496	**36**	6	**36**	4	**36**	22	17	299	4
rovers	22	22	3	22	< 1	22	< 1	22	< 1	22	1	1
satellite	20	20	22	20	< 1	20	< 1	20	< 1	20	259	1
difficult-contest05-jarvisalo	10	**3**	323	2	659	0	-	0	-	0	-	1
medium-contest05-jarvisalo	10	2	782	**9**	1955	1	470	3	850	3	497	4
spence-medium	10	3	220	**9**	1031	4	294	4	111	8	877	5
grieu	10	**5**	1222	1	71	**5**	347	3	525	**5**	378	1
industrial-jarvisalo	7	1	16	3	112	3	400	2	99	**4**	448	5

TABLE 1 – Les résultats de DPLL-TD et comparaisons

çon d'élaborer un tel ordre en calculant les composantes biconnexes du graphe de contraintes modélisant le problème traité. Par ailleurs, des travaux plus récents ont montré que l'usage de la décomposition arborescente peut guider avec succès l'heuristique de branchement utilisée par DPLL et nous présentons ici quelques-uns de ces travaux. L'heuristique de choix de variable présentée dans [9] utilise la méthode Dtree [4], qui fournit une décomposition sous forme d'un arbre binaire et statique, pour ordonner les variables par groupes. L'arbre de décomposition est construit avant la résolution par DPLL. De ce fait, l'ordre des groupes de variables ne change jamais au cours de l'exécution. Toutefois, l'ordre global établi n'est ni statique, ni dynamique, car si l'ordre des groupes est statique, l'ordre des variables à l'intérieur même de ces groupes reste dynamique. Dans [3], les auteurs présentent une heuristique pour créer dynamiquement un ordre sur des groupes de variables. Le bémol majeur de cette approche est qu'aucun résultat expérimental n'est présenté pour montrer si leur méthode est plus efficace qu'un ordre statique, comme c'est le cas dans le travail précédent. En outre, il n'existe aucune conclusion sur la façon d'organiser les sous-problèmes induits par la décomposition arborescente. Dans [2], les auteurs proposent une heuristique qui permet de résoudre d'abord les problèmes les plus contraints. Mais là encore, aucune évaluation expérimentale ne vient appuyer leur approche. Une heuristique d'ordre des variables basée sur une méthode récursive de type *min-cut bisection* de l'hypergraphe représentant l'instance a été proposée dans [1]. Cette approche ne nécessite pas la modification du solveur SAT utilisé. Cependant, la quasi-totalité des solveurs SAT modernes utilise un ordre de variables dynamique. L'ordre statique fourni par cette dernière approche sert notamment à remplacer, dans certaines situations, l'ordre dynamique établi par le solveur SAT. Dans [12], les auteurs décrivent une méthode de décomposition dynamique (sous forme d'un arbre) basée sur les séparateurs de l'hypergraphe représentant l'instance SAT traitée. L'usage de ces séparateurs dans l'établissement de l'ordre de choix de variables de branchement permet, sur quelques instances, d'accélérer les temps de réponse de certains solveurs SAT modernes. Comparée à Dtree, cette méthode ne nécessite pas de temps pour la construction complète de l'arbre de décomposition. Cette construction peut exiger, dans certains cas, plus de temps que la résolution effective de l'instance elle-même. Par ailleurs, dans [12], il est présenté un schéma de partitionnement contraint qui peut être exploité pour dériver un ordre sur les variables pour accélérer les solveurs SAT. Cette approche, qui est à l'opposé des schémas de décomposition arborescente qui cherchent à réduire la largeur de la décomposition arborescence, élabore un partitionnement sous contraintes de l'hypergraphe représentant l'instance SAT par l'analyse du nombre d'occurrences des variables dans les clauses de l'instance SAT et la connectivité existante entre ces clauses. Cette connectivité est exprimée par les variables communes à ces clauses.

Enfin, la méthode DPLL-TD a un lien de parenté évident avec la méthode BTD [10] proposée dans le cadre des problèmes de satisfaction de contraintes. Si ces deux méthodes reposent sur la notion de décomposition arborescente et son exploitation pour guider la recherche tout en mémorisant des *(no)goods* structurels, elles n'en demeurent pas moins différentes. D'une part, si les problèmes SAT et CSP sont voisins, leurs méthodes de résolution ne mettent pas en œuvre exactement les mêmes techniques. Par exemple, si toutes les variables doivent être instanciées pour produire une solution d'un CSP, il n'est pas nécessaire de toutes les instancier pour trouver un modèle d'une instance SAT. Cette simple différence rend nécessaire la définition d'un cadre formel spécifique au problème SAT pour l'al-

gorithme DPLL-TD (en particulier, avec l'introduction de la notion de variable d'indépendance). D'autre part, la notion de *good* diffère significativement entre DPLL-TD et BTD. De plus, les *goods* et *nogoods* de DPLL-TD peuvent être de taille variable en ne portant pas sur l'ensemble des variables du séparateur. Il en résulte ainsi des coupes potentiellement plus puissantes pour DPLL-TD que celles utilisées dans BTD.

6 Conclusion et discussions

Dans ce papier, nous avons proposé une nouvelle approche basée sur la décomposition arborescente pour résoudre le problème de satisfiabilité, notamment les instances structurées issues des problèmes du monde réel.

Le but de cette approche est de capturer la structure de l'instance SAT traitée. Cette structure est décrite sous la forme d'un hypergraphe qui est donc décomposé. La décomposition arborescente ainsi exploitée fournit un ordre sur les variables qui est utilisé pour guider une méthode énumérative de type DPLL. De plus et durant la recherche, des *goods* et *nogoods* structurels sont appris et utilisés pour élaguer l'espace de recherche. Cet apprentissage constitue l'une des originalités de notre approche, notamment en comparaison avec celles décrites dans la section précédente. Un autre apport de DPLL-TD est de fournir des bornes de complexité théorique, en temps et en espace, qui dépendent de la décomposition arborescente.

Contrairement au cadre des problèmes de satisfaction de contraintes (CSP), les approches basées sur la notion de décomposition arborescente sont peu exploitées dans le cadre SAT. Comme en témoigne la section précédente à nouveau, un nombre non négligeable de travaux s'inscrivant dans cette approche ne fournissent pas de résultats expérimentaux, ce qui rend difficilement mesurable leurs efficacités et contributions pratiques. Par conséquent, on pourrait dire que la littérature existante manque de maturité et de recul sur l'usage de la décomposition arborescente pour SAT. En effet, plusieurs aspects devraient être étudiés pour renforcer le travail que nous présentons ici. Parmi ces aspects, on peut citer la méthode de triangulation utilisée qui est nécessaire pour le calcul de la décomposition arborescente, le choix du cluster racine, l'élaboration de plusieurs heuristiques de choix du prochain cluster à traiter, ... Comme premières pistes, nous travaillerons sur l'usage de l'analyse des conflits afin d'apprendre de nouveaux nogoods à l'instar de ce qui fait dans les solveurs basés sur CDCL, l'usage des restarts tout en modifiant soit le cluster racine ou même la décomposition elle-même, à la base des informations apprises lors des précédentes exécutions. Par exemple, construire une nouvelle décomposition arborescente sur la base des variables les plus conflictuelles (contraintes). Cette information peut-être également exploitée à guider l'ordre de parcours des clusters en s'intéressant aux plus contraints d'abord.

Références

[1] F.A. Aloul, I.L. Markov, and K.A. Sakallah. MINCE : A Static Global Variable-Ordering Heuristic for SAT Search and BDD Manipulation. *Journal of Universal Computer Science (JUCS)*, 10(12) :1562–1596, 2004.

[2] E. Amir and S. Mcilraith. Solving satisfiability using decomposition and the most constrained subproblem. In *LICS workshop on Theory and Applications of Satisfiability Testing*, 2001.

[3] P. Bjesse, J. Kukula, R. Damiano, T. Stanion, and Y. Zhu. Guiding sat diagnosis with tree decompositions. In *Proceedings of SAT 2004*, pages 315–329, 2004.

[4] A. Darwiche. A compiler for deterministic, decomposable negation normal form. In *Proceedings of AAAI 2002*, pages 627–634, 2002.

[5] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7) :394–397, 1962.

[6] N. Eén and N. Sörensson. An extensible sat-solver. In *Proceedings of SAT 2003*, pages 402–518, 2003.

[7] N. Eén and N. Sörensson. Effective preprocessing in sat through variable and clause elimination. In *Proceedings of SAT 2005*, pages 61–75, 2005.

[8] E.C. Freuder. A sufficient condition for backtrack-bounded search. *J. ACM*, 32(4) :755–761, 1985.

[9] J. Huang and A. Darwiche. A structure-based variable ordering heuristic for sat. In *Proceedings of IJCAI 2003*, pages 1167–1172, 2003.

[10] P. Jégou and C. Terrioux. Hybrid backtracking bounded by tree-decomposition of constraint networks. *Artificial Intelligence*, 146 :43–75, 2003.

[11] C.M. Li and Anbulagan. Heuristic Based on Unit Propagation for Satisfiability. In *Proceedings of CP'97*, pages 342–356, Austria, 1997.

[12] W. Li and P. van Beek. Guiding real-world sat solving with dynamic hypergraph separator decomposition. In *Proceedings of ICTAI 2004*, pages 542–548, Washington, DC, USA, 2004. IEEE Computer Society.

[13] K. Pipatsrisawat and A. Darwiche. Rsat 2.0 : Sat solver description. Technical Report D–153, Automated Reasoning Group, Computer Science Department, UCLA, 2007.

[14] N. Robertson and P.D. Seymour. Graph minors II : Algorithmic aspects of treewidth. *Algorithms*, 7 :309–322, 1986.

[15] U. Kjærulff. Triangulation of graphs : Algorithms giving small total state space. Technical report, University of Aalborg, 1990.

[16] L. Zhang and C.F. Madigan. Efficient conflict driven learning in a boolean satisfiability solver. In *Proceedings of ICCAD 2001*, pages 279–285, 2001.

Localiser des sources d'incohérence spécifiques sans les calculer toutes

Éric Grégoire **Bertrand Mazure** **Cédric Piette**

Université Lille-Nord de France, Artois, F-62307 Lens
CRIL, F-62307 Lens
CNRS UMR 8188, F-62307 Lens

{gregoire,mazure,piette}@cril.fr

Résumé

Cet article pose le problème suivant : étant donné un sous-ensemble Γ d'une formule CNF Σ, Γ prend-il part aux sources de l'incohérence de Σ ? Pour répondre à cette question, une approche originale permettant de vérifier si Γ chevauche au moins une sous-formule minimale incohérente (MUS) de Σ est présentée. De plus, si la réponse est positive, un tel MUS est calculé et retourné. Cette technique ré-exprime le problème dans un cadre où les clauses de Σ sont regroupées en *clusters* qui sont manipulés comme les entités atomiques de la formule. Les clusters sont ensuite progressivement affinés selon leur niveau de conflits mutuels, et les plus prometteurs sont sélectionnés et divisés, permettant d'élaguer les clauses inutiles jusqu'à ce que la quantité de ressources allouées soit épuisée ou jusqu'à ce qu'une solution soit retournée. La viabilité et l'utilité de cette approche sont évaluées empiriquement.

1 Introduction

Ces dernières années, de nombreuses études se sont concentrées sur l'explication de l'incohérence d'une instance SAT, fournie à travers des ensembles minimaux de clauses conflictuelles. En effet, bien que certains solveurs (e.g. [5, 1]) puissent fournir une trace de la preuve d'incohérence calculée, celle-ci n'est pas toujours suffisante car elle ne garantit pas que l'ensemble de clauses retourné devienne cohérent avec la suppression de n'importe lequel de ses membres. Par conséquent, plusieurs contributions récentes ont porté sur la localisation de sous-ensembles minimalement incohérents de clauses (MUS pour *Minimal Unsatisfiable Subformula*) d'instances SAT incohérentes ([11, 21], consulter [10] pour une vue d'ensemble récente des différentes techniques). En terme de complexité dans le pire des cas, le calcul des MUS posent plusieurs problèmes majeurs. Tout d'abord, une formule CNF contenant n clauses peut posséder jusqu'à $C_n^{n/2}$ MUS dans le pire cas. Ensuite, vérifier si une formule donnée appartient ou non à l'ensemble des MUS d'une CNF Σ est un problème Σ_2^p-difficile [4]. Toutefois, des approches pour localiser un MUS viables dans de nombreuses circonstances ont été proposées [22, 9]. En outre, dans le but de circonvenir à la possible explosion combinatoire du nombre de MUS d'une formule CNF, des variantes au problème de l'extraction exhaustive des MUS ont été proposée, comme celui de concept de couverture inconsistante, représentant un ensemble de causes minimales et non corrélées de l'incohérence [9]. Enfin, des algorithmes pour calculer l'ensemble complet des MUS de Σ ont également été proposés et se montrent efficaces, au moins pour certains problèmes ne possédant pas un trop grand nombre de MUS.

Quand un utilisateur est face à une instance SAT incohérente Σ, il peut avoir certaines croyances à propos d'un sous-ensemble de clauses Γ potentiellement en cause dans l'un des conflits de Σ. Dans ce cadre, il peut désirer vérifier si ses croyances sont fondées et si Γ participe effectivement à l'incohérence de Σ. En dehors de la situation où $\Sigma \setminus \Gamma$ est satisfiable pour laquelle la réponse est évidente, c'est une requête qui est calculatoirement lourde, tout particulièrement si l'utilisateur désire obtenir une explication sous forme d'un MUS contenant des informations de Γ. En fait, les techniques actuelles ne permettent pas de répondre à une requête comme celle-ci sans calculer l'ensemble des MUS de Σ.

Nous proposons dans ce papier une approche origi-

nale pour répondre à ce problème, sans calculer *tous* les MUS de Σ. Pour circonvenir à la très haute complexité dans le pire cas (du moins, dans une certaine mesure) de cette requête, le problème est exprimé dans un cadre à gros grains où des clusters de clauses de Σ sont formés, puis considérés comme les éléments de base de la formule et examinés selon leur niveau de conflits mutuels. Le cadre est ensuite progressivement raffiné en divisant les clusters les plus prometteurs et en élaguant ceux qui ont été prouvés inutiles jusqu'à ce que la quantité de ressources allouées soit épuisée ou jusqu'à ce qu'une solution soit retournée. Notons que le niveau de conflits mutuels entre clusters est mesuré grâce à une forme de valeur de Shapley [20], célèbre dans la communauté de la théorie des jeux.

Le papier est organisé comme suit. Dans la section suivante, les préliminaires formels sont donnés, ainsi que les définitions de base, certaines propriétés sur les MUS et la définition de la mesure d'incohérence de Shapley. Les principes généraux de l'approche sont décrits en section 3. Le schéma de notre algorithme est détaillé en section 4, tandis que des résultats expérimentaux sont fournis et discutés en section 5. Certains travaux connexes sont ensuite décrits avant de conclure par des pistes de recherche que nous pensons prometteuses.

2 Préliminaires logiques, MUS et mesure de Shapley

2.1 Préliminaires logiques et SAT

Soit $\mathcal{L}$ le langage propositionnel standard, défini inductivement de manière usuelle à partir d'un ensemble P de symboles propositionnels (représentés par des lettres minuscules a, b, c, etc.), les constantes booléennes $\top$ et $\bot$, et les connecteurs standard $\neg$, $\wedge$, $\vee$, $\Rightarrow$ et $\Leftrightarrow$. Une instance SAT est une formule propositionelle sous forme CNF, c'est-à-dire une conjonction (représentée par un ensemble) de clauses, où une clause est une disjonction (représentée par un ensemble) de littéraux, un littéral étant une variable propositionelle ou sa négation. Dans la suite, les lettres grecques majuscules comme Δ, Γ, etc. seront utilisées pour représenter des ensembles de clauses, les lettres grecques minuscules telles que α, β, γ etc. pour représenter des clauses.

SAT est le problème de décision NP-complet qui consiste à vérifier si une CNF est satisfiable, c'est-à-dire si elle admet au moins un modèle. Par la suite, nous utiliserons indifféremment les termes satisfiable (resp. insatisfiable) et cohérent (resp. incohérent). En outre, tout au long de ce papier, nous supposons que Σ est une instance SAT incohérente et Γ est un ensemble de clauses tel que $\Gamma \subset \Sigma$.

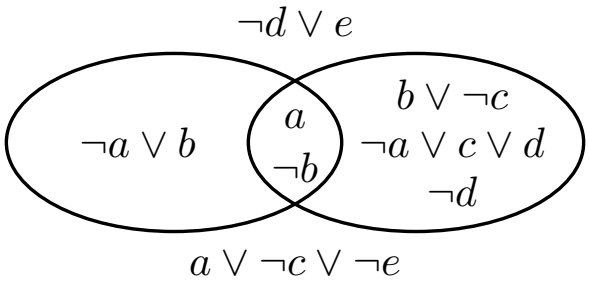


FIG. 1 – MUS de l'Exemple 1.

2.2 MUS d'une instance SAT

Un MUS d'une instance SAT insatisfiable Σ est un sous-ensemble de clauses de Σ qui ne peut être rendu plus petit sans restaurer sa cohérence. Un MUS représente par conséquent une cause minimale de l'incohérence, exprimée sous forme de clauses conflictuelles.

Définition 1 *Un ensemble de clauses Γ est un sous-ensemble minimalement incohérent (MUS pour* Minimally Unsatisfiable Subformula*) de Σ ssi :*

1. $\Gamma \subseteq \Sigma$
2. Γ *est insatisfiable*
3. $\forall \Delta \subset \Gamma$, Δ *est satisfiable.*

L'exemple suivant illustre le fait que deux MUS peuvent se chevaucher l'un l'autre.

Exemple 1 *Soit* $\Sigma = \{\neg d \vee e,\ b \vee \neg c,\ \neg d,\ \neg a \vee b,\ a,\ a \vee \neg c \vee \neg e,\ \neg a \vee c \vee d,\ \neg b\}$. Σ *est insatisfiable et contient 2 MUS* $\{a,\ \neg a \vee b,\ \neg b\}$ *et* $\{b \vee \neg c,\ \neg d,\ a,\ \neg a \vee c \vee d,\ \neg b\}$ *représentés en Figure 1.*

Cet exemple illustre également les différents rôles que peut jouer une clause par rapport à l'incohérence d'une instance SAT Σ. Suivant la catégorisation proposée dans [16], une clause est dite *nécessaire* si celle-ci apparaît dans tous les MUS de Σ. Supprimer une telle clause de Σ en restaure la satisfiabilité. Les clauses *potentiellement nécessaires* de Σ appartiennent à au moins l'un de ses MUS, mais pas à tous. Retirer l'une de ces clauses ne suffit pas à restaurer la cohérence de l'instance SAT, mais elles peuvent devenir nécessaires avec la suppression de clauses appropriées. Les clauses n'appartenant à aucune de ces 2 catégories (c'est-à-dire n'appartenant à aucun MUS) sont appelées *non nécessaires*. Retirer n'importe quel combinaison de ces clauses ne permet en aucun cas de restaurer la satisfiabilité de Σ. Assez

clairement, une instance SAT incohérente ne contient pas toujours de clauses nécessaires, puisque celle-ci peut contenir 2 MUS ne se chevauchant pas.

Exemple 1 (suite) *Les 2 clauses « a » et « $\neg b$ » sont nécessaires par rapport Σ, « $\neg d \vee e$ » et « $a \vee \neg c \vee \neg e$ » sont non nécessaires, tandis que les autres clauses de Σ sont potentiellement nécessaires.*

De nombreuses techniques ont été proposées pour la localisation d'un MUS au sein d'une CNF incohérente [10]. Cependant, ces approches ne peuvent *a priori* pas tenir compte de conditions stipulant que le MUS calculé doit contenir certaines clauses. En conséquence, dans le but de calculer une source de conflits spécifique, la seule solution consiste en l'utilisation d'outils tels que [17, 8] qui retourne l'ensemble complet des MUS de Σ. Notons que les plus efficaces de ces approches extraient les sous-formules maximales satisfiables (MSS pour *Maximal Satisfiable Subsets*) de Σ [17] de la formule comme étape préliminaire, et en dérivent par ensemble intersectant celui des MUS. Ainsi, ces techniques sont difficilement viables en pratique quand des formules de grande taille sont considérées, même si le nombre de MUS de celles-ci est relativement petit par rapport au nombre de clauses de la CNF.

2.3 Mesure d'incohérence de Shapley

Il existe de nombreuses études visant à mesurer les différents niveaux d'incohérence d'un ensemble de formules. Dans la suite de ce papier, nous utiliserons une forme de mesure de Shapley [20] présentée dans [12]. La mesure d'incohérence de Shapley d'une clause α dans une instance SAT Σ fournit un score basé sur le nombre de MUS de Σ contenant α, ainsi que sur la taille de chacun de ces MUS, ce qui permet de tenir également compte de « l'importance » de chaque clause dans chaque source d'incohérence.

Définition 2 *[12] Soient Σ une instance SAT et $\alpha \in \Sigma$. Soit $\cup_{MUS_\Sigma}$ l'ensemble des MUS de Σ. La mesure d'incohérence MI de α dans Σ, notée $MI(\Sigma, \alpha)$, est définie comme :*

$$MI(\Sigma, \alpha) = \sum_{\{\Delta t.q. \Delta \in \cup_{MUS_\Sigma} \ et \ \alpha \in \Delta\}} \frac{1}{|\Delta|}$$

Les propriétés d'une telle mesure sont étudiées dans [12]. L'idée générale est qu'une clause apparaissant dans de nombreux conflits va obtenir un score important, alors qu'une clause contenue dans un « grand » MUS aura un plus faible score qu'une autre clause contenue dans un MUS plus petit.

Exemple 2 *En considérant la CNF Σ de l'Exemple 1, on a :*
- $MI(\Sigma, \neg a \vee b) = \frac{1}{3}$
- $MI(\Sigma, b \vee \neg c) = \frac{1}{5}$
- $MI(\Sigma, \neg d \vee e) = 0$
- $MI(\Sigma, a) = MI(\Sigma, \neg b) = \frac{1}{3} + \frac{1}{5} = \frac{8}{15}$.

Dans la suite de ce papier, cette mesure sera étendue pour caractériser les niveaux de conflits entre des formules conjonctives.

3 Principes généraux de l'approche

3.1 Définition du problème

On dit qu'un ensemble de clauses booléennes Γ *participe effectivement* à l'incohérence d'une instance SAT Σ si Γ contient au moins une clause qui appartient à au moins un MUS de Σ, c'est-à-dire une clause qui est nécessaire ou potentiellement nécessaire par rapport à l'incohérence de Σ. Dans ce cas, notre but est de fournir l'un des MUS de Σ chevauchant Γ. Plus formellement :

Définition 3 *Soient $\cup_{MUS_\Sigma}$ l'ensemble des MUS d'un ensemble de clauses booléennes Σ et Γ un sous-ensemble de Σ. Γ participe à l'incohérence de Σ ssi $\exists \alpha \in \Gamma, \exists \Delta \in \cup_{MUS_\Sigma}$ t.q. $\alpha \in \Delta$.*

De manière évidente, si Γ contient au moins une clause nécessaire (par rapport à Σ), alors elle participe effectivement à son incohérence. Dans ce cas spécifique, vérifier sa participation ne nécessite au plus que k appels à un solveur SAT, où k est le nombre de clauses de Γ, puisqu'il existe $\alpha \in \Sigma \cap \Gamma$ t.q. $\Sigma \setminus \{\alpha\}$ soit satisfiable. De plus, dans ce cas tout MUS de Σ chevauche Γ et fournit une solution à notre problème. Cependant, dans le cas général, Γ ne contient pas nécessairement une telle clause, rendant le problème du calcul d'un MUS contenant l'une de ces clauses particulières plus difficile.

3.2 Utilisation des clusters pour simplifier le problème

Pour circonvenir à la forte complexité du problème de l'extraction d'un MUS spécifique, nous avons recours à une stratégie consistant à partitioner Σ en un nombre prédéfini de m sous-ensembles, appelés clusters, comme étape préliminaire.

Définition 4 *Soit Σ une formule CNF. Un m-*clustering de clauses Π *de Σ est une partition de Σ en m sous-ensembles $\Pi_j (j \in [1..m])$. Π_j est appelé le $j^{ème}$ cluster de Σ (par rapport au clustering Π de Σ).*

Pour des raisons pratiques, nous utiliserons la notation Π_j pour représenter à la fois l'ensemble des clauses formant le $j^{ème}$ cluster de Σ et la formule faite de la conjonction de ces clauses. De plus, l'union ensembliste de clusters représentera la conjonction des clauses qu'ils contiennent.

Les clauses contenues au sein d'un même cluster sont considérées conjonctivement pour devenir une formule vue comme une entité indivisible au sein de Σ. Les clauses de Γ sont traitées de la même manière. Le problème initial est donc élevé à un niveau d'abstraction moins fin, consistant à tester la façon dont la formule Γ prend place dans l'incohérence de Σ, maintenant composée de m sous-formules atomiques.

3.3 Des MUS aux MUSC

Le problème que nous traitons ici est donc relégué à un cadre où l'incohérence est analysée à travers des clusters de clauses. Le concept de MUS doit donc être adapté en conséquence, donnant naissance au concept d'ensemble minimalement incohérent de clusters pour un clustering Π (noté $MUSC_\Pi$ pour *Minimally Unsatisfiable Set of Clusters*) de Σ.

Définition 5 *Soient Σ une formule CNF et Π un m-clustering de clauses de Σ. Un* ensemble minimalement incohérent de clusters *pour Π (noté $MUSC_\Pi$) M_Π de Σ est un ensemble de clusters t.q. :*

1. $M_\Pi \subseteq \Pi$
2. M_Π *est insatisfiable*
3. $\forall \Phi \in M_\Pi$, $M_\Pi \setminus \{\Phi\}$ *est satisfiable*

L'ensemble des $MUSC_\Pi$ de Σ est noté $\cup_{MUSC_\Pi}$.

Le concept « classique » de MUS est une instanciation particulière de celui de MUSC, où chaque cluster est en fait une simple clause. Par conséquent, les MUS représentent le niveau d'abstraction le plus fin pour expliquer l'incohérence de Σ, tandis que celui de MUSC permet la découverte d'explication de l'incohérence à gros grains.

Considérons les clauses dont la participation à l'incohérence d'une instance SAT Σ veut être établie. Quand de telles clauses apparaissent dans un $MUSC_\Pi$ de Σ (pour n'importe quel clustering Π), elles participent également à l'incohérence de Σ quand le problème est traité à des niveaux d'abstraction inférieurs, c'est-à-dire au niveau des clauses. Plus formellement :

Proposition 1 *Soient Σ une instance SAT, Π un m-clustering de Σ et M_Π un $MUSC_\Pi$ de Σ. Si $\Pi_i \in M_\Pi$, alors $\exists \alpha \in \Pi_i, \exists \Psi \in \cup_{MUS_\Sigma}$ t.q. $\alpha \in \Psi$.*

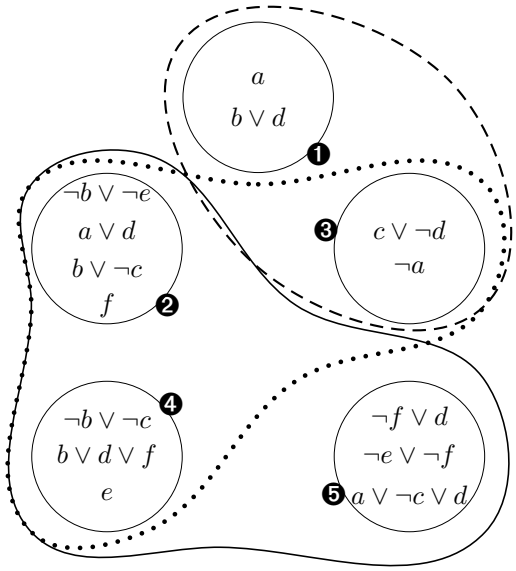


FIG. 2 – MUS de la CNF Σ de l'Exemple 3.

Une telle transformation de ce problème possède des caractéristiques intéressantes, d'un point de vue calculatoire. Celles-ci seront exploitées dans notre approche. Cependant il n'y a pas de miracle : comme le problème est une simplification du problème initial, certaines sources d'incohérence peuvent disparaître. Toutefois, cette simplification reste tout à fait compatible avec notre but initial et un algorithme correct et complet (dans le sens où il retourne un MUS de Σ chevauchant Γ s'il en existe un) est proposé.

Exemple 3 *Soit Σ une instance SAT composée de 14 clauses, telle qu'illustrée en Figure 2. Ces 14 clauses forment 9 MUS. Par exemple, prouver que les clauses « $\neg a$ » et « $c \vee \neg d$ » participent effectivement à l'incohérence de Σ nécessite dans le pire cas l'extraction de l'ensemble exhaustif des MUS de cette CNF. Regroupons maintenant les clauses en 5 clusters, de la façon dépeinte en Figure 2. À ce niveau d'abstraction, on ne compte que 3 $MUSC_\Pi$, qui sont $M_1 = \{❶,❸\}$, $M_2 = \{❷,❸,❹\}$ et $M_3 = \{❷,❹,❺\}$, en utilisant les étiquettes des clusters de la Figure. Ce clustering limite donc le nombre d'entités en contradiction tout en prouvant qu'au moins l'une des clauses « $\neg a$ » ou « $c \vee \neg d$ » appartient à un MUS de Σ.*

Cet exemple illustre l'intérêt de grouper les clauses en clusters, permettant d'éviter le calcul de *tous* les MUS pour répondre à la requête de l'existence d'un MUS spécifique. Cependant, si certaines clauses groupées en un certain clustering n'apparaissent dans aucun MUSC, ceci ne signifie pas nécessairement qu'elles ne participent pas à l'incohérence de la CNF au niveau des clauses.

Proposition 2 *Soient Σ une instance SAT et Π un n-clustering de Σ. Si $\exists \Pi_i \in \Pi, \exists$ une clause $\alpha \in \Pi_i, \exists$ un MUS $\Psi \in MUS_\Sigma$ t.q. $\alpha \in \Psi$, alors Π_i n'apparaît pas nécessairement dans un $MUSC_\Pi$ de Σ.*

Exemple 4 *Soit $\Sigma = \{\neg c, \neg a, b \vee c, a, \neg b\}$. Σ possède 2 MUS : $MUS^1_\Sigma = \{\neg a, a\}$ et $MUS^2_\Sigma = \{\neg b, b \vee c, \neg c\}$. Considérons maintenant le 3-clustering Π de Σ t.q. $\Pi_1 = \{\neg b, a\}$, $\Pi_2 = \{\neg a, b \vee c\}$ et $\Pi_3 = \{\neg c\}$. Nous obtenons :*

	Π_1	Π_2	Π_3
MUS^1_Σ	a	$\neg a$	
MUS^2_Σ	$\neg b$	$b \vee c$	$\neg c$

À ce niveau d'abstraction, le seul $MUSC_\Pi$ est $\{\Pi_1, \Pi_2\}$, capturant MUS^1_Σ. Cependant, MUS^2_Σ n'est pas représenté dans cet unique $MUSC_\Pi$, puisque ses clauses sont divisées dans chaque cluster de Π. En particulier, ce clustering ne permet pas de conclure quant à la participation effective de « $\neg c$ » dans l'incohérence de Σ.

Il existe donc un phénomène de « subsumption » entre les causes de l'incohérence quand les clauses sont regroupées en clusters. D'un côté, le nombre de sources d'incohérence de Σ décroît en conséquence (ce qui est notre but) et une approche adaptée peut tirer parti des MUSC pour se focaliser sur les plus prometteurs. D'un autre côté, il faut s'assurer que l'information qui nous intéresse sur l'incohérence de Σ ne soit pas cachée par le clustering, rendant notre but de calculer un MUS spécifique impossible à atteindre, bien qu'un tel MUS existe. Dans une certaine mesure, cet exemple illustre également que la façon dont les clusters sont formés (et les MUS disséminés en leur sein) influe de manière cruciale sur l'efficacité d'une telle approche.

3.4 Mesurer les conflits grâce aux MUSC

La mesure d'incohérence de Shapley décrite en section 2.3 nécessite d'être adaptée aux clusters de clauses, maintenant considérés comme éléments atomiques d'une instance SAT.

Définition 6 *Soient Σ une instance SAT et Π un m-clustering de Σ. La mesure d'incohérence MI d'un cluster Π_i de Π est définie comme suit :*

$$MI(\Pi_i) = \sum_{\{M | M \in \cup_{MUSC_\Pi} \text{ et } \Pi_i \in M\}} \frac{1}{|M|}$$

Une fois l'ensemble des MUSC extrait, cette mesure peut être obtenue en temps polynomial.

Exemple 5 *Soient Σ l'instance SAT et Π le clustering de l'Exemple 3. On a :*

- $MI(\Pi_1) = 1/2$
- $MI(\Pi_2) = MI(\Pi_4) = 2 \times 1/3 = 2/3$
- $MI(\Pi_3) = 1/2 + 1/3 = 5/6$
- $MI(\Pi_5) = 1/3$

Le cluster possédant le score le plus important est Π_3. Ceci est en partie dû au fait que Π_3 appartient à 2 $MUSC_\Pi$. Π_2 et Π_4 appartiennent également à 2 $MUSC_\Pi$, mais chacun d'eux ne représente que le tiers des 2 sources d'incohérence, alors que Π_3 est impliqué dans un $MUSC_\Pi$ composé de seulement 2 clusters. Le rôle de Π_3 est donc important dans ce conflit, et son score est pondéré en conséquence.

3.5 Initialiser le processus

Dans le but de former un clustering initial, une heuristique peu coûteuse en ressources est utilisée. Celle-ci fournit un score à chaque clause de Σ en fonction de sa probabilité d'appartenance à au moins un MUS de la CNF. Cette heuristique est basée sur une recherche locale et le concept de *clause critique* qui permet de prendre en compte un voisinage partiel pertinent de chaque interprétation parcourue [9].

Σ est divisé en m clusters de la manière suivante. Les $\frac{|\Sigma \setminus \Gamma|}{m}$ clauses de $\Sigma \setminus \Gamma$ qui ont les scores les plus importants prennent (probablement) part à un grand nombre de conflits et leur présence dans Σ peut être la cause de nombreux MUS. Celles-ci sont donc rassemblées pour former un cluster Π_1. Ensuite, les autres clauses de $\Sigma \setminus \Gamma$ sont également triées par rapport à leur score et groupées de la même manière, pour obtenir les $m-1$ clusters de Π et fournir un m-clustering de $\Sigma \setminus \Gamma$. Les clauses de Γ sont alors ajoutées pour avoir un $(m+1)^{ème}$ cluster, noté Π_Γ. En conséquence, un $m+1$-clustering de Σ est obtenu.

3.6 Analyser les conflits au niveau des clusters

Chaque cluster est donc interprété comme une formule conjonctive et l'intéraction entre ces entités est analysée en terme (d'ensembles) de clusters mutuellement contradictoires. Cette analyse de conflits a pour but d'élaguer le problème en rejetant les clusters qui ne sont pas contradictoires avec Γ, et en se concentrant sur ceux qui sont au contraire les plus en conflit avec Γ. L'ensemble exhaustif des MUSC de ce clustering (i.e. $\cup_{MUSC_\Pi}$) est donc calculé et on est alors face à l'une des trois situations suivantes :

1. *Incohérence globale :* une forme d'incohérence globale survient au niveau des clusters quand pour chaque Π_i ($i \in [1..m]$) du m-clustering Π, $\Pi \setminus \Pi_i$ est cohérent. Une telle situation se produit quand au moins une clause de chaque cluster est nécessaire pour former un conflit. Le clustering ne

fournit alors aucune information réellement pertinente. Le paramètre m (représentant le nombre de clusters) est alors incrémenté pour obtenir un clustering plus fin. Notons que cette situation ne requiert pas énormémement de ressources pour être vérifiée, puisque que ce clustering ne contient qu'un seul MUSC et un nombre linéraire d'ensembles satisfiables de clusters.

2. Γ *apparaît dans au moins conflit :* certains conflits locaux ont été détectés entre les clusters, et au moins l'un d'entre eux contient Π_Γ. Plus précisément, $\exists M \in \cup_{MUSC_\Pi}$ t.q. $\Pi_\Gamma \in M$. Dans ce cas, l'idée est de se concentrer sur un seul MUSC puisqu'il implique que certaines clauses de Γ sont nécessairement en conflit avec d'autres, contenues dans les clusters de ce MUSC. En conséquence, l'ensemble courant de clauses peut être élagué en supprimant toutes les clauses n'apparaissant pas dans ce MUSC particulier contenant Π_Γ.

3. Γ *n'apparaît dans aucun conflit :* certains conflits entre clusters ont été identifiés mais aucun d'entre eux ne comprend le cluster composé des clauses de Γ. Plus précisément, $\nexists M \in \cup_{MUSC_\Pi}$ t.q. $\Pi_\Gamma \in M$. Dans une telle situation, un raffinement apparaît nécessaire. Plutôt qu'un redécoupage général (cf. situation d'incohérence globale), nous pouvons ici bénéficier d'informations importantes sur les sources de conflits. En effet, les sources de conflits étant ici locales, il nous est possible de sélectionner les clusters les *plus conflictuels* en utilisant la mesure de Shapley décrite précédemment. Ainsi, un nombre prédéterminé de clusters ayant les scores de Shapley les plus importants sont divisés, selon l'intuition que ceux-ci cachent de nombreuses autres sources d'incohérence, dans l'espoir d'en découvrir une impliquant Γ.

3.7 Itération et fin du processus

Le clustering est donc modifié selon l'une des 3 situations présentées ci-dessus, permettant d'augmenter le nombre de clusters, les diviser et se concentrer sur les sous-ensembles de Σ les plus prometteurs. Comme ce processus est itéré, nous obtenons en temps fini un $|\Sigma|$-clustering (chaque cluster est en fait une seule clause), correspondant au calcul « classique » de MUS, sur un nombre réduit de clauses de Σ. En pratique, quand la taille du MUSC comprenant Γ devient de taille raisonnable pour envisager un calcul exhaustif, celui-ci est effectué. L'un des attraits de cette approche est son caractère *any-time* : après qu'une certaine quantité de ressources soit épuisée, celle-ci peut fournir la solution courante, c'est-à-dire la sous-formule de Σ courante, en conflict avec Γ.

La technique que nous proposons, basée sur ces idées, est décrite dans la section suivante et est illustrée dans l'algorithme 1.

4 Algorithme principal

Dans cette section, nous décrivons l'algorithme principal de l'approche proposée. Pour des raisons pratiques, certains cas spécifiques gérés dans notre implémentation (par exemple les situations où Γ est incohérent ou Σ est cohérent) ne sont pas décrites. Notre présentation n'inclut pas non plus le prétraitement syntaxique qui consiste à supprimer les occurences de clauses identiques dans Σ, sauf une (celle appartenant à Γ si une telle clause existe), permettant de réduire exponentiellement le nombre de MUS de la CNF. Il est important de traiter de telles situations pour éviter des cas pathologiques, mais celles-ci sont simples à implémenter et ne présentent que peu d'intérêt algorithmique. L'algorithme se réfère également à la technique `HYCAM` [8] qui calcule l'ensemble des MUS d'une instance SAT. En pratique, une nouvelle version de `HYCAM` capable de gérer les clusters de clauses a été implémentée et utilisée dans les études empiriques.

Détaillons les différentes étapes de l'approche décrite dans l'Algorithme 1. Cette technique prend en entrée une CNF Σ et l'un de ses sous-ensembles Γ. De plus, elle nécessite l'entrée de 3 paramètres entiers qui sont m, inc et s.

Tout d'abord, une recherche locale est exécutée pour fournir un score heuristique aux clauses suivant la contrainte que chacune exerce au sein de Σ. Plus précisément, les scores de chaque clause sont tout d'abord mis à 0. Durant le processus de recherche locale, les scores des clauses *critiques* [9] par rapport à l'interprétation courante sont incrémentés (cf. fonction `scoreClause` pour plus de détails).

Les clauses sont ensuite groupées en un clustering Π (ligne 3), via l'appel de la fonction `clusterClauses`. Cette fonction consiste à faire la conjonction des clauses ayant les scores les plus importants dans un premier cluster, les clauses restantes ayant les scores maximum dans un deuxième cluster, etc. jusqu'à ce que m clusters soient formés. Puis, la version modifiée de `HYCAM`, notée `HYCAM*` (ligne 4) est appelée pour calculer l'ensemble des MUSC de Π.

À présent, l'ensemble $\cup_{MUSC_\Pi}$ a été calculé et plusieurs cas sont à considérer. Soit Π est globalement incohérent (lignes 9-10) : à ce niveau, aucune information ne peut être extraite pour trouver un sous-ensemble de clauses conflictuelles impliquant Γ, le clustering est donc raffiné en ajoutant inc clusters supplémentaires (dans l'algorithme, ceci est effectué à travers un appel récursif à `look4MUS`). Dans le cas contraire, nous

Algorithme 1 : L'algorithme `look4MUS`.

Données :
Σ : une CNF
Γ : une CNF t.q. $\Gamma \subset \Sigma$
m : taille du clustering initial
inc : incrément du clustering (incohérence globale)
s : nombre de clusters à diviser (incohérence locale)
Résultat : Un MUS de Σ chevauchant Γ si un tel MUS existe, sinon $\emptyset$

```
1  début
2      S_Σ ⟵ scoreClauses(Σ) ;
3      Π ⟵ clusterClauses(Σ \ Γ, S_Σ, m) ∪{Γ} ;
4      ∪MUSC_Π ⟵ HYCAM*(Π) ;
5      si Π est un |Σ|-clustering alors
6          si ∃M ∈ ∪MUSC_Π t.q. Γ ⊆ M alors retourner M ;
7          sinon retourner ∅ ;
8      sinon
9          si ∀Π_i ∈ Π, Σ \ Π_i est satisfiable alors
               // incohérence globale
10             retourner look4MUS(Σ, Γ, m + inc, inc, s) ;
11         sinon
12             tant que ∄M ∈ ∪MUSC_Π t.q. Γ ∈ M faire
                   // incohérence locale
13                 Π' ⟵ ∅ ;
14                 pour j ∈ [1..s] faire
15                     Π_best ⟵ ∅ ;
16                     pour chaque Π_i ∈ Π t.q. Π ≠ Γ faire
17                         si MI(Π_i,∪MUSC_Π)>MI(Π_best,∪MUSC_Π) alors Π_best ⟵ Π_i ;
18                     Π ⟵ Π \ Π_best ;
19                     Π' ⟵ Π' ∪ clusterClauses(Π_best, S_Σ, 2) ;
20                 Π ⟵ Π ∪ Π' ;
21                 ∪MUSC_Π ⟵ HYCAM*(Π) ;
22             M_Γ ⟵ selectOneMUSC(∪MUSC_Π,Γ) ;
23             retourner look4MUS(M_Γ, Γ, m, inc, s) ;
24 fin
```

sommes en présence d'une incohérence locale. Tant que Γ n'est impliqué dans aucun $MUSC_\Pi$ (lignes 12-21), les s clusters les plus conflictuels (par rapport à la mesure d'incohérence de Shapley de la fonction `MI`) sont divisés en deux parties dans le but de révèler une source d'incohérence impliquant Γ. Quand de telles incohérences locales sont découvertes, un MUSC M_Γ contenant Γ est sélectionné (fonction `selectOneMUSC`, qui retourne le MUSC contenant le moins de clauses possible). Le processus continue en ne considérant que la CNF M_Γ, qui est une sous-formule de Σ qui contient un MUS impliquant Γ.

À moins que le temps de calcul imparti soit épuisé, l'algorithme ne peut stopper qu'aux lignes 6 et 7, quand chaque cluster est en fait une simple clause ($|\Sigma|$-clustering). À ce point, un appel « classique » à `HYCAM` est effectué, fournissant des MUS (les « MUSC » ligne 6 sont en fait des MUS puisque chaque cluster est une clause) contenant de l'information de Γ, ou prouvant que de tels MUS n'existe pas. Dans notre implémentation, nous n'attendons pas nécessairement qu'un $|\Sigma|$-clustering soit atteint pour utiliser la procédure classique ; quand le nombre de clusters est suffisamment proche du nombre de clauses (i.e. quand $m < 2 \times |\Sigma|$), alors la procédure se termine par un appel à `HYCAM`. De plus, l'implémentation possède une caractéristique *any-time*, dans le sens où elle fournit la dernière sous-formule calculée quand la quantité de ressources allouées à cette tâche est épuisée.

Fonction `scoreClauses`

Données : Σ : une CNF
Résultat : Un vecteur de scores pour chaque clause de Σ

début
  pour chaque $c \in \Sigma$ **faire**
    $S_\Sigma(c) \longleftarrow 0$;
  $I \longleftarrow$ une affectation aléatoire pour chaque variable de Σ ;
  tant que *le nombre de flip imparti n'est pas dépassé* **faire**
    pour chaque $c \in \Sigma$ **faire**
      si *c est critique p/r à I dans* Σ **alors**
        $S_\Sigma(c)++$;
    $I \longleftarrow I'$ t.q. I et I' diffèrent d'une seule valeur de vérité ;
  retourner S_Σ ;
fin

Fonction `clusterClauses`

Données : Σ : une CNF, S_Σ : un vecteur de scores, m : la taille du clustering
Résultat : Un m-clustering de Σ

début
  $\Pi \longleftarrow \emptyset$;
  pour $i \in [1..m-1]$ **faire**
    $\Pi_i \longleftarrow$ les $(\frac{|\Sigma|}{m})$ clauses de Σ ayant les plus hauts scores ;
    $\Pi \longleftarrow \Pi \cup \{\Pi_i\}$;
    $\Sigma \longleftarrow \Sigma \setminus \Pi_i$;
  $\Pi \longleftarrow \Pi \cup \{\Sigma\}$;
  retourner Π ;
fin

Fonction `selectOneMUSC`

Données : $\cup_{MUSC_\Pi}$: l'ensemble des MUSC p/r Π, Γ : une CNF
Résultat : Un MUSC de $\cup_{MUSC_\Pi}$ contenant Γ

début
  $M_\Gamma \longleftarrow \Pi$;
  pour chaque M *t.q.* $M \in \cup_{MUSC_\Pi}$ **faire**
    si $\Gamma \subseteq M$ *et* $|M| < |M_\Gamma|$ **alors**
      $M_\Gamma \longleftarrow M$;
  retourner M_Γ ;
fin

Fonction `MI`

Données : Π_i : un élément du clustering Π, $\cup_{MUSC_\Pi}$: l'ensemble des $MUSC_\Pi$
Résultat : La mesure d'incohérence de Π_i p/r Π

début
  $mi \longleftarrow 0$;
  pour chaque $MUSC_\Pi \in \cup_{MUSC_\Pi}$ **faire**
    si $\Pi_i \in MUSC_\Pi$ **alors**
      $mi \longleftarrow mi + \frac{1}{|MUSC_\Pi|}$;
  retourner mi ;
fin

5 Résultats expérimentaux

Une implémentation en `C` de l'algorithme présenté a été réalisée. Comme cas d'étude, les valeurs suivantes ont été sélectionnées comme paramètres $m = 30$, $inc = 10$ et $s = \frac{m}{10}$. Cette implémentation a été exécutée sur de nombreux benchmarks provenant de `http://www.satcompetition.org`. Toutes nos expérimentations ont été conduites sur des processeurs Intel Xeon 3GHz sous Linux CentOS 4.1. (kernel 2.6.9) avec une limite de mémoire RAM de 2 Go.

Pour chaque instance SAT testée, nous essayons de prouver que les clauses apparaissant en $1^{ère}$, $3^{ème}$, $5^{ème}$ et $7^{ème}$ position de son fichier au format DIMACS participent effectivement à l'incohérence de la CNF, et d'extraire un MUS contenant au moins l'une de ces clauses. Un échantillon des résultats obtenus est reporté en Table 1 avec pour chaque instance, la taille en nombre de clauses du MUS extrait ($\#\text{cla}_{MUS}$) et le temps en secondes. Ces résultats permettent de constater que l'approche est capable de localiser des MUS spécifiques pour des CNF de taille conséquente. Notons de plus que l'ensemble exhaustif des MSS (et donc des MUS) ne peut être extrait en temps raisonnable ni par `CAMUS` [17] ni par `HYCAM` [8], pour la plupart de ces problèmes. Aucune approche jusqu'à présent ne permet donc d'extraire une source d'incohérence contenant exactement telle ou telle information. Notre technique permet au contraire de répondre à cette question de la participation de clauses spécifiques dans l'incohérence des formules testées.

Grouper les clauses en cluster dans le but de « dissimuler » des sources d'incohérence, et se concentrer sur les clusters impliquant les clauses désirées se montre particulièrement viable en pratique. En effet, cette stratégie permet de réduire de manière drastique l'effort calculatoire en limitant l'explosion combinatoire du nombre de sources d'incohérence. Par exemple, les problèmes `dp05u04` et `dp06u05` (qui encodent le célèbre problème du dîner des philosophes dans le cadre du *bounded model checking*) possède un si grand nombre de MUS qu'il est impossible de les calculer

nom	#var	#cla	#cla$_{MUS}$	temps
1dlx_...bp_f	776	3725	1440	705
bf0432-007	1040	3668	1223	119
bf1355-075	2180	6778	151	22
bf1355-638	2177	6768	154	21
bf2670-001	1393	3434	134	8
dp05u04	1571	3902	820	1254
dp06u05	2359	6053	1105	2508
ezfact16_1	193	1113	319	28
ezfact16_2	193	1113	365	43
ezfact16_3	193	1113	297	31
ezfact16_10	193	1113	415	54

TAB. 1 – Extraire un MUS of Σ chevauchant Γ : quelques résultats expérimentaux

tous (voire même de les lister) avec la technologie actuelle. `look4MUS` réussit au contraire à extraire l'une des sources d'incohérence voulues – sans les extraire toutes – en un temps de calcul très raisonnable (environ respectivement 20 et 42 minutes). La situation est similaire pour les autres CNF. Sur la famille d'instances `ezfact16_*` (factorisation de circuits), une explication impliquant des clauses particulières peut être extraite en moins d'une 1 minute alors que les approches exhaustives montrent leurs limites sur de tels problèmes. Néanmoins, les problèmes considérés dans la Table 1 sont loin des problèmes de très grande taille (plusieurs dizaines de milliers de clauses) que peuvent résoudre les meilleurs solveurs SAT actuels. Notons tout de même que sur certains de ces problèmes difficiles, notre technique est parvenue à itérer plusieurs fois, permettant de retourner en temps raisonnable une sous-formule conflictuelle avec Γ, grâce à son caractère *any-time*.

6 Travaux connexes

Les approches existantes pour extraire un MUS ou en faire l'approximation ont été décrites dans les sections précédentes. Soulignons ici que le travail présenté dans ce papier induit une forme d'*abstraction* pour réduire le coût calculatoire du problème initial. De plus, via l'aspect *any-time* de cette technique, elle peut être utilisée comme une stratégie d'*approximation*. De la même manière, un grand nombre de techniques d'abstraction et d'approximation ont été proposées pour traiter divers problèmes liés à la logique propositionelle. Nous présentons ici quelques unes d'entre elles.

Le papier séminal de l'approximation sous ressources limitées de l'inférence en logique propositionelle est [19]. Une autre approximation, où chaque étape de calcul peut être décidée en temps polynomial, a été proposée dans le cadre clausal dans [2, 3], et plus tard étendue à l'ensemble de la logique classique dans [6]. Pour un survol des techniques d'approximation, nous renvoyons le lecteur à [7]. Notons enfin que l'approximation de raisonnement dans les systèmes à base de connaissances a également été un sujet très étudié (voir par exemple [14, 13]).

Pour finir, l'idée de grouper les clauses en clusters a déjà été utilisée dans le contexte de contraintes de haut niveau codée sous forme de clauses [17]. De plus, il apparaît que les cas où aucune clause de Γ ne participe à l'incohérence sont les pires pour notre approche. Toutefois, si Γ est composée exclusivement de clauses non nécessaires, alors cette sous-formule peut être satisfaite par une *autarky* [15], qui est définie comme une interprétation partielle qui satisfait toutes les clauses ayant au moins un littéral affecté. Un algorithme a récemment été proposé pour calculer des autarkys en modifiant la CNF et en considérant un problème d'optimisation [18].

7 Conclusions et travaux futurs

Dans ce papier, une technique originale permettant de vérifier si un ensemble de clauses chevauche au moins une sous-formule minimale incohérente d'une formule CNF a été décrite. Un tel problème peut se montrer très utile quand l'utilisateur désire restaurer la cohérence d'une telle formule et possède (ou peut obtenir) certaines informations à propos des clauses en conflit de la CNF.

L'approche proposée se montre plus efficace que les approches existantes de plusieurs ordres de grandeur, grâce à son paradigme d'abstraction et de reformulation. Elle ne nécessite en particulier pas l'extraction de *tous* les MUS de l'instance. Un autre attrait de la technique est son caractère *any-time*, permettant de produire des solutions plus grossières si les ressources qui lui sont allouées sont limitées.

Les pistes de recherche liées à cette approche incluent la prise en compte de plusieurs variantes du cadre de travail. Par exemple, l'un des points cruciaux de l'approche repose sur la façon dont les clauses sont regroupées en clusters. Bien que notre heuristique basée sur la recherche locale et le concept de clause critique se soit montrée très fructueuse en pratique, il pourrait être intéressant de réfléchir à de nouveaux regroupements tenant compte de différentes formes d'interaction entre les clauses. De plus, cette étude se concentre sur le cas booléen classique, où chaque clause a la même importance que les autres. L'utilisateur peut cependant avoir certaines préférences qualitative ou quantitative à propos des informations contenues dans Σ. Dans ce cas, il préfèrera extraire le MUSC

qui correspond le mieux à ses préférences. Nous prévoyons d'étudier l'extension de notre approche à ce cadre.

Références

[1] Armin Biere. Boolforce. `http://fmv.jku.at/booleforce`.

[2] Mukesh Dalal. Anytime families of tractable propositional reasoners. In *AI/MATH'96*, pages 42–45, 1996.

[3] Mukesh Dalal. Semantics of an anytime family of reasponers. In *ECAI'96*, pages 360–364, 1996.

[4] Thomas Eiter and Georg Gottlob. On the complexity of propositional knowledge base revision, updates and counterfactual. *Artificial Intelligence*, 57 :227–270, 1992.

[5] Niklas Eén and Niklas Sörensson. Minisat home page. `http://www.cs.chalmers.se/Cs/Research/FormalMethods/MiniSat`.

[6] Marcelo Finger. Towards polynomial approximations of full propositional logic. In *SBIA'04*, pages 11–20. LNAI, 2004.

[7] Marcelo Finger and Renata Wassermann. The universe of propositional approximations. *Theoretical Computer Science*, 355(2) :153–166, 2006.

[8] Éric Grégoire, Bertrand Mazure, and Cédric Piette. Boosting a complete technique to find MSS and MUS thanks to a local search oracle. In *IJCAI'07*, volume 2, pages 2300–2305. AAAI Press, 2007.

[9] Éric Grégoire, Bertrand Mazure, and Cédric Piette. Local-search extraction of MUSes. *Constraints Journal*, 12(3) :325–344, 2007.

[10] Éric Grégoire, Bertrand Mazure, and Cédric Piette. On approaches to explaining infeasibility of sets of Boolean clauses. In *ICTAI'08*, volume 1, pages 74–83, 2008.

[11] Jinbo Huang. MUP : A minimal unsatisfiability prover. In *ASP-DAC'05*, pages 432–437, 2005.

[12] Anthony Hunter and Sébastien Konieczny. Measuring inconsistency through minimal inconsistent sets. In *KR'08*, pages 358–366, 2008.

[13] Frédéric Koriche. Approximate coherence-based reasoning. *Journal of Applied Non-Classical Logics*, 12(2) :239–258, 2002.

[14] Frédéric Koriche and Jean Sallantin. A logical toolbox for knowledge approximation. In *TARK'01*, pages 193–205. Morgan Kaufmann Publishers Inc., 2001.

[15] Oliver Kullmann. Investigations on autark assignments. *Discrete Applied Mathematics*, 107 :99–137, 2000.

[16] Oliver Kullmann, Ines Lynce, and Joao Marques Silva. Categorisation of clauses in conjunctive normal forms : Minimally unsatisfiable sub-clause-sets and the lean kernel. In *SAT'06*, pages 22–35, 2006.

[17] Mark Liffiton and Karem Sakallah. Algorithms for computing minimal unsatisfiable clause sets. *Journal of Automated Reasoning*, 40(1) :1–33, 2008.

[18] Mark Liffiton and Karem Sakallah. Searching for autarkies to trim unsatisfiable clause sets. In *SAT'08*, pages 182–195, 2008.

[19] Marco Schaerf and Marco Cadoli. Tractable reasoning via approximation. *Artificial Intelligence*, 74 :249–310, 1995.

[20] Lloyd Shapley. A value for n-person games. *Contributions to the Theory of Games II (Annals of Mathematics Studies 28)*, pages 307–317, 1953.

[21] Hans van Maaren and Siert Wieringa. Finding guaranteed MUSes fast. In *SAT'08*, pages 291–304, 2008.

[22] Lintao Zhang and Sharad Malik. Extracting small unsatisfiable cores from unsatisfiable Boolean formula. In *SAT'03*, 2003.

Subsumption dirigée par l'analyse de conflits

Youssef Hamadi[1] **Said Jabbour**[2] **Lakhdar Sais**[2]

[1] Microsoft Research
7 J J Thomson Avenue
Cambridge, United Kingdom
youssefh@microsoft.com

[2] CRIL-CNRS, Université d'Artois
rue Jean Souvraz SP18
F-62307 Lens Cedex France
{jabbour,sais}@cril.fr

Abstract

Cet travail présente une technique originale de subsumption pour les formules booléennes sous forme normale conjonctive (CNF). Il exploite une condition simple et suffisante pour détecter, durant l'analyse de conflits, les clauses de la formule qui peuvent être réduites par subsumption. Durant le processus de dérivation de la clause à apprendre, et à chaque étape du processus de résolution associé, le test de subsumption entre la résolvante courante et une clause de la formule originale ou apprise auparavant est alors efficacement effectué. La méthode résultante permet d'éliminer dynamiquement des littéraux appartenant aux clauses de la formule courante. Les résultats expérimentaux montrent que l'intégration de notre technique de subsumption dynamique dans les solveurs `Minisat` et `Rsat` est bénéfique, et cela particulièrement sur les problèmes dits "crafted".

1 Introduction

Le problème SAT, i.e., la vérification de la satisfaction d'un ensemble de clauses, est central dans plusieurs domaines et notamment en intelligence artificielle incluant le problème de satisfaction de contraintes (CSP), planification, raisonnement non-monotone, vérification de logiciels, etc. Aujourd'hui, SAT a gagné une audience considérable avec l'apparition d'une nouvelle génération de solveurs SAT capable de résoudre de très grandes instances ainsi que par la le fait que ces solveurs constituent d'importants composants de base pour plusieurs domaines, e.g., SMT(SAT modulo théorie), preuve de théorème, comptage de modèles, problème QBF, etc. Ces solveurs, appelés solveurs SAT modernes [13, 9], sont basés sur la propagation de contraintes classique [6] efficacement combinée à d'efficaces structures de données avec : (i) stratégies de redémarrage [10, 11], (ii) heuristiques de choix de variables (comme VSIDS) [13], et (iii) apprentissage de clauses [1, 12, 13]. Les solveurs SAT modernes peuvent être vus comme une extension de la procédure DPLL classique obtenues grâce à différentes améliorations. Il est important de noter que la règle de résolution basique à la Robinson joue encore un rôle prépondérant dans l'efficacité des solveurs modernes qui peut être vu comme une forme particulière de la résolution générale [2].

En effet, l'apprentissage basé sur les conflits, l'un des composants importants des solveurs SAT est basé sur la résolution. On peut mentionner aussi que l'un des prétraitement le plus connu et dont le succès est largement répandu (`SatElite`) est basé sur l'élimination de variables par l'application de la règle de résolution [16, 3]. Comme cité dans [16], sur les instances industrielles, la résolution génère plusieurs résolvantes tautologiques. Ceci peut être expliqué par le faite que plusieurs clauses codant des fonctions booléennes sont reliées par un ensemble de variables communes. Cette propriété du codage peut être à l'origine de plusieurs clauses redondantes ou subsumées aux difèrentes étapes du processus de recherche.

L'utilité de (`SatElite`) sur les problèmes industriels étant prouvée, l'on est en droit de se demander si l'application de la règle de résolution pourrait être réalisée non seulement comme une étape de pré-traitement mais systématiquement, pendant le processus de recherche. Malheureusement, maintenir une formule fermée par subsumption peut être très coûteux. Une tentative a été faite récemment dans cette direction par L. Zhang [17]. Dans ce travail, un nouvel algorithme maintient la base de clauses fermée par subsumption en supprimant dynamiquement les clauses subsumées au moment de leur ajout. Plus intéressant, l'auteur indique cette perspective de recherche : "trouver un équilibre entre le coût d'exécution et la qualité de la simplification par subsumption à la volée d'une CNF est un problème qui mérite plus d'intérêt et d'investigation".

Dans cet article, notre objectif est de concevoir un algorithme de subsumption dynamique simple et efficace basé sur la résolution. L'approche proposée a pour but d'éliminer des littéraux de la formule CNF en substituant dynamiquement les clauses originales par des clauses plus courtes. Plus précisemment, nous exploitons les étapes intermidiares de l'analyse de conflits classique pour subsumer des clauses de la formule qui sont utilisées dans la processus de résolution relatif à la génération de la clause assertive. Comme les clauses originales ou les clauses aprisent peuvent être utilisées durant l'analyse de conflits les deux catégories peuvent être simplifiées.

L'efficacité de notre technique réside dans la simplicité du test de subsumption, qui est basé sur une condition simple et suffisante calculable en temps constant. En plus, du faite que notre technique repose sur le processus de génération de la clause conflit, elle est aussi guidée par les conflits, et simplifie la partie de la formule identifiée comme étant importante par la stratégie de recherche (dirigé par VSIDS). Ce processus dynamique préserve la satisfiabilité de la formule, et avec quelques règles de compatibilité additionnelles peut préserver l'équivalence des modèles.

Notre article est organisé comme suit. Après quelques notations et définitions préliminaires, le graphe d'implications classique et le schéma d'apprentissage sont présentés dans la section 2. Ensuite notre approche de subsumption dynamique est décrite dans la section 3. Finalement, avant la conclusion, des résultats expérimentaux démontrent les performances de notre approche sur les catégories d'instances testées.

2 Definitions

2.1 Définitions et notations préliminaires

Une *formule CNF* $\mathcal{F}$ est une conjonction de *clauses*, où une clause est une disjonction de *littéraux*. Un littéral est interprété comme une positif (x) ou negatif ($\neg x$) propositional variable. Les deux littéraux x et $\neg x$ sont appelés *complémentaire*. On note par $\bar{l}$ le littéral complémentaire de l. Pour un ensemble de littéraux L, $\bar{L}$ est définit comme $\{\bar{l} \mid l \in L\}$. Une clause *unitaire* est une clause contenant seulement un seul littéral (appelé *littéral unitaire*), tandis qu'une clause binaire contient exactement deux littéraux. Une *clause vide*, notée $\bot$, est interprété comme fausse (unsatisfiable), alors qu'une *formule CNF vide*, notée $\top$, est interprétée comme vraie (satisfiable).

L'ensemble des variables apparaissant dans $\mathcal{F}$ est noté $V_{\mathcal{F}}$. Un ensemble de littéraux est *complet* s'il contient un littéral pour chaque variable de $V_{\mathcal{F}}$, et *fondamental* s'il contient pas de littéraux complémentaites. Une *affectation* ρ d'une formule booléenne $\mathcal{F}$ est une fonction qui associe la valeur $\rho(x) \in \{vrai, faux\}$ à quelques variables de $x \in \mathcal{F}$. ρ est *complète* s'il attribue une valeur pour chaque variable $x \in \mathcal{F}$, et *partielle* sinon. Une affectation est alternativement représentée par un ensemble de littéraux complet et fondamentale, de façon évidente un *modèle* d'une formule $\mathcal{F}$ est une affectation ρ qui rend la formule $vraie$; notée $\rho \vDash \mathcal{F}$.

Les notations suivantes sont largement utilisées le reste de l'article.

- $\eta[x, c_i, c_j]$ dénote la *résolvante* entre une clause c_i contenant un littéral x et c_j une autre clause contenant l'opposé de ce même littéral $\neg x$.
  En d'autres termes, $\eta[x, c_i, c_j] = c_i \cup c_j \backslash \{x, \neg x\}$. Une résolvante est appelée *tautologique* s'elle contient à la fois un littéral et son opposé.
- $\mathcal{F}|_x$ représente la formule obtenu à partir de $\mathcal{F}$ en affectant à x la value de vérité *vrai*. Formellement $\mathcal{F}|_x = \{c \mid c \in \mathcal{F}, \{x, \neg x\} \cap c = \emptyset\} \cup \{c \backslash \{\neg x\} \mid c \in \mathcal{F}, \neg x \in c\}$ (les clauses contenant x (satisfaites) sont supprimées ; et celles contenant $\neg x$ sont simplifiées). Cette notation est étendue aux affectations : étant donnée une affectation $\rho = \{x_1, \ldots, x_n\}$, on définit $\mathcal{F}|_\rho = (\ldots((\mathcal{F}|_{x_1})|_{x_2})\ldots|_{x_n})$.
- $\mathcal{F}^*$ dénote la formule $\mathcal{F}$ fermée par propagation unitaire, définit récursivement comme suit : (1) $\mathcal{F}^* = \mathcal{F}$ si $\mathcal{F}$ ne contient pas des clauses unitaires, (2) $\mathcal{F}^* = \bot$ si $\mathcal{F}$ contient deux clauses unitaires $\{x\}$ et $\{\neg x\}$, (3) sinon, $\mathcal{F}^* = (\mathcal{F}|_x)^*$ tel que x est le littéral apparaissant dans une clause unitaire de $\mathcal{F}$. Une clause c est déduite par propagation unitaire à partir de $\mathcal{F}$, notée $\mathcal{F} \models^* c$, si $(\mathcal{F}|_{\bar{c}})^* = \bot$.

Soient c_1 et c_2 deux clauses d'une formule $\mathcal{F}$. On dit que c_1 (respectivement c_2) subsume (respectivement est subsumée) c_2 (respectivement par c_1) ssi $c_1 \subseteq c_2$. Si c_1 subsume c_2, alors $c_1 \models c_2$ (l'inverse est faux). Aussi $\mathcal{F}$ et $\mathcal{F} - c_2$ sont équivalentes pour la satisfiabilité.

2.2 Recherche DPLL

Nous introduisons maintenant des notations et remarques terminologiques associées aux solveurs SAT basés sur la procédure de Davis Logemann Loveland, communément appelé DPLL [6]. DPLL est une procédure de recherche de type "*backtrack*" ; À chaque nœud les littéraux affectés (le littéral de décision et les littéraux propagés) sont étiquetés avec le même *niveau de décision*, initialisé à 1 et incrémenté à chaque nouveau point de décision. Le niveau de décision courant est le niveau le plus élevé dans la pile de propagation. Lors d'un retour-arrière ("*backtrack*"), les variables ayant un niveau supérieur au niveau du backtrack sont défaites et le niveau de décision courant est décrémenté en consèquence (égal au niveau du backtrack). Au niveau i, l'interprétation partielle courante ρ peut être représentée comme une séquence décision–propagations de la forme $\langle (x_k^i), x_{k_1}^i, x_{k_2}^i, \ldots, x_{k_{n_k}}^i \rangle$ telle que le premier lit-

téral x_k^i correspond au littéral de décision x_k affecté au niveau i et chaque $x_{k_j}^i$ de l'ensemble $1 \leq j \leq n_k$ représente les litéraux unitaires propagés à ce même niveau i.

Soit $x \in \rho$, On note $l(x)$ le niveau d'affectation de x, $d(\rho, i) = x$ si x est le littéral de décision affecté au niveau i.

2.3 Analyse de conflit et graphe d'implications

Le graphe d'implications est une représentation standard utilisée classiquement pour analyser les conflits dans les solveurs SAT mdernes. À chaque fois qu'un littéral y est propagé, on sauvegarde une référence de la clause impliquant la propagation de y, qu'on note $\overrightarrow{cla}(y)$. La clause $\overrightarrow{cla}(y)$, appelée implication de y, est dans ce cas de la forme $(x_1 \vee \cdots \vee x_n \vee y)$ où chaque littéral x_i est faux sous l'affectation partielle courante ($\rho(x_i) = \textit{faux}, \forall i \in 1..n$), alors que $\rho(y) = \textit{vrai}$. Lorsqu'un littéral y n'est pas obtenu par propagation mais issue d'une décision, $\overrightarrow{cla}(y)$ est indéfinie, qu'on note par convention $\overrightarrow{cla}(y) = \perp$. Quand $\overrightarrow{cla}(y) \neq \perp$, on note par $exp(y)$ l'ensemble $\{\overline{x} \mid x \in \overrightarrow{cla}(y) \setminus \{y\}\}$, appelé ensemble des *explications* de y. Alors que $\overrightarrow{cla}(y)$ est indéfini on défini $exp(y)$ comme l'ensemble vide.

Définition 1 (Graphe d'implications) *Soit $\mathcal{F}$ une formule CNF, ρ une affectation partielle, et soit exp l'ensemble des explications pour les littéraux déduits (propagé unitairement) dans ρ. Le graphe d'implications associé à $\mathcal{F}$, ρ et exp est $\mathcal{G}_{\mathcal{F}}^{\rho,exp} = (\mathcal{N}, \mathcal{E})$ où :*

- *$\mathcal{N} = \rho$, i.e., il existe un nœud pour chaque littéral de décision ou propagé ;*
- *$\mathcal{E} = \{(x, y) \mid x \in \rho, y \in \rho, x \in exp(y)\}$*

Dans le reste du papier, pour des raisons de simplicité, exp est éliminée, et un graphe d'impliacations est tout simplement noté $\mathcal{G}_{\mathcal{F}}^{\rho}$. On note par m le niveau du conflit.

Exemple 1 *$\mathcal{G}_{\mathcal{F}}^{\rho}$, montré dans la Figure 1 représente le graphe d'implications de la formule $\mathcal{F}$ et l'affectation partiel ρ donnée ci-dessous : $\mathcal{F} \supseteq \{c_1, \ldots, c_{12}\}$*

$(c_1) \neg x_1 \vee \neg x_{11} \vee x_2$	$(c_2) \neg x_1 \vee x_3$
$(c_3) \neg x_2 \vee \neg x_{12} \vee x_4$	$(c_4) \neg x_1 \vee \vee \neg x_3 \vee x_5$
$(c_5) \neg x_4 \vee \neg x_5 \vee \neg x_6 \vee x_7$	$(c_6) \neg x_5 \vee \neg x_6 \vee x_8$
$(c_7) \neg x_7 \vee x_9$	$(c_8) \neg x_5 \vee \neg x_8 \vee \neg x_9$
$(c_9) \neg x_{10} \vee \neg x_{17} \vee x_1$	$(c_{10}) \neg x_{13} \vee \neg x_{14} \vee x_{10}$
$(c_{11}) \neg x_{13} \vee x_{17}$	$(c_{12}) \neg x_{15} \vee \neg x_{16} \vee x_{13}$

$\rho = \{\langle \ldots x_{15}^1 \ldots \rangle \langle (x_{11}^2) \ldots \ldots \rangle \langle (x_{12}^3) \ldots x_6^3 \ldots \rangle \langle (x_{14}^4) , \ldots \rangle \langle (x_{16}^5), x_{13}^5, \ldots \rangle\}$. Le niveau du conflit est 5 et $\rho(\mathcal{F}) = faux$.

Définition 2 (Clause assertive) *Une clause c de la forme $(\alpha \vee x)$ est appelée clause assertive ssi $\rho(c) = \textit{faux}$, $l(x) = m$ et $\forall y \in \alpha, l(y) < l(x)$. x est appelé littéral assertif.*

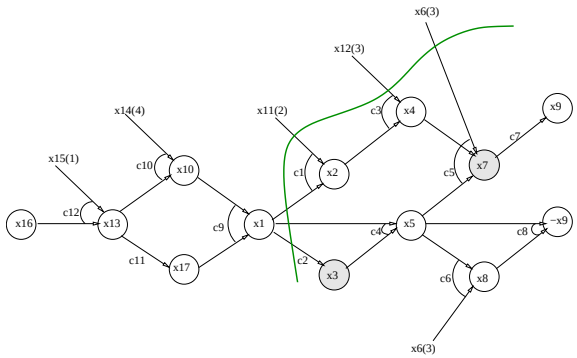


FIG. 1 – Graphe d'implications $\mathcal{G}_{\mathcal{F}}^{\rho} = (\mathcal{N}, \mathcal{E})$

L'analyse du conflit est le résultat de l'application de la résolution à partir de la clause conflit en utilisant différentes implications implicitement codées dans le graphe d'implications. On appelle ce processus dérivation de la clause assertive (DCA en court).

Définition 3 (dérivation de la clause assertive) *Le dérivation de la clause assertive π est une séquence de clauses $\langle \sigma_1, \sigma_2, \ldots \sigma_k \rangle$ satisfaisant les conditions suivantes :*

1. *$\sigma_1 = \eta[x, \overrightarrow{cla}(x), \overrightarrow{cla}(\neg x)]$, tel que $\{x, \neg x\}$ est le conflit.*
2. *σ_i, pour $i \in 2..k$, est construite en sélectionnant un littéral $y \in \sigma_{i-1}$ pour lequel $\overrightarrow{cla}(\overline{y})$ est définit. Nous avons alors $y \in \sigma_{i-1}$ et $\overline{y} \in \overrightarrow{cla}(\overline{y})$: appliquer la résolution entre ces deux clauses mène à la clause σ_i qui est définit comme $\eta[y, \sigma_{i-1}, \overrightarrow{cla}(\overline{y})]$;*
3. *σ_k est en plus une clause assertive.*

Notons que chaque σ_i est une résolvante de la formule $\mathcal{F}$: par induction, σ_1 est la résolvante entre deux clauses qui appartiennent directement à $\mathcal{F}$; pour chaque $i > 1$, σ_i est une résolvante entre σ_{i-1} (qui est, par l'hypothèse d'induction, une résolvante) et une clause de $\mathcal{F}$. Chaque σ_i est également un *impliqué* de $\mathcal{F}$, ce qui veut dire que : $\mathcal{F} \models \sigma_i$. Par définition du graphe d'implication, on a aussi $\mathcal{F}' \models^* \sigma_i$ tel que $\mathcal{F}' \subset \mathcal{F}$ est l'ensemble des clauses utilisées pour générer σ_i.

Considérons à nouveau l'exemple 1. Le parcours du graphe $\mathcal{G}_{\mathcal{F}}^{\rho}$ (voir Fig. 1) est décrit par la dérivation de la clause assertive suivant :

$\langle \sigma_1, \ldots, \sigma_7 \rangle$ tel que $\sigma_1 = \eta[x_9, c_7, c_8] = (\neg {x_5}^5 \vee \neg {x_7}^5 \vee \neg {x_8}^5)$ et $\sigma_7 = (\neg {x_{11}}^2 \vee \neg {x_{12}}^3 \vee \neg {x_6}^3 \vee \neg {x_1}^5)$. Le nœud x_1 correspond au littéral assertif, $\neg x_1$ est appelé le premier Unique Point d'Implication (First UIP).

3 Apprentissage et subsumption dynamique

Dans cette section, nous montrons comment l'apprentissage classique peut être adapté pour une subsumption dynamique et efficace des clauses. Dans notre approche, on exploite les différentes clauses générées pendant les étapes de résolution du processus d'analyse de conflit. Bien sûr, il est possible de considérer la subsumption entre chaque résolvante générée et l'ensemble entier des clauses. Cependant, celà peut être coûteux en pratique. Illustrons certaines des principales caractéristiques de notre approche proposée.

3.1 Exemple

Considérons à nouveau l'exemple 1 et le graphe d'implications $\mathcal{G}^{\rho}_{\mathcal{F}}$ (figure 1). Le dérivation de la clause assertive permettant de déduire la clause Δ_1 est décrit comme suit :
$\pi = \langle \sigma_1, \sigma_2, \sigma_3, \ldots, \sigma_7 = \Delta_1 \rangle$
- $\sigma_1 = \eta[x_9, c_7, c_8] = (\neg {x_8}^5 \vee \neg {x_7}^5 \vee \neg {x_5}^5)$
- $\sigma_2 = \eta[x_8, \sigma_1, c_6] = (\neg {x_6}^3 \vee \neg {x_7}^5 \vee \neg {x_5}^5)$
- $\sigma_3 = \eta[x_7, \sigma_2, c_5] = (\neg {x_6}^3 \vee \neg {x_5}^5 \vee \neg {x_4}^5) \subset c_5$ (subsumption)
- $\ldots$
- $\Delta_1 = \sigma_7 = \eta[x_2, \sigma_6, c_1] = (\neg {x_{11}}^2 \vee \neg {x_{12}}^3 \vee \neg {x_6}^3 \vee \neg {x_1}^5)$

Comme on peut le voir, ce processus π inclut la résolvante $\sigma_3 = (\neg {x_6}^3 \vee \neg {x_5}^5 \vee \neg {x_4}^5)$ qui subsumes la clause $c_5 = (\neg x_6 \vee \neg x_5 \vee \neg x_4 \vee x_7)$. Par conséquent, le littéral x_7 peut être éliminé de la clause c_5. En général, la résolvante σ_3 peut subsumer d'autres clauses du graphe d'implications qui contiennent le littéral $\neg x_6$, $\neg x_5$ et $\neg x_4$.

3.2 Subsumption dynamique : formulation générale

On donne à présent une présentation formelle de notre approche basée sur la subsumption dynamique.

Définition 4 (F-subsumption modulo PU) *Soit $c \in \mathcal{F}$. c est $\mathcal{F}$-subsumée modulo la propagation unitaire ssi $\exists c' \subset c$ tel que $\mathcal{F}|_{\vec{c'}} \models^* \bot$.*

Soient c_1 et c_2 deux clauses de $\mathcal{F}$ telles que c_1 subsumes c_2, alors c_2 est $\mathcal{F}$-subsumed modulo PU.

Comme expliqué auparavant, la subsumption des clauses durant la recherche peut être coûteuse. Dans le cadre proposé, pour réduire le coût calculatoire, on retreint la vérification de la subsumption aux résolvantes intermidiares σ_i et les clauses de la forme $\overrightarrow{cla}(y)$ utilisées pour les générer (clauses codées dans le graphe d'implications).

Définition 5 *Soit $\mathcal{F}$ une formule et $\pi = \langle \sigma_1 \ldots \sigma_k \rangle$ le dérivation de la clause assertive . Pour chaque $\sigma_i \in \pi$, on définit $\mathcal{C}_{\sigma_i} = \{\overrightarrow{cla}(y) \in \mathcal{F} | \exists j \leq i$ tq. $\sigma_j = \eta[y, \overrightarrow{cla}(y), \sigma_{j-1}]\}$ comme l'ensemble de clauses de $\mathcal{F}$ utilisé dans la déduction de σ_i.*

Propriété 1 *Soit $\mathcal{F}$ une formule et $\pi = \langle \sigma_1, \sigma_2, \ldots, \sigma_i, \ldots, \sigma_k \rangle$ le dérivation de la clause assertive . Si σ_i subsume une clause c de $\mathcal{C}_{\sigma_k}$ alors $c \in \mathcal{C}_{\sigma_i}$.*

Preuve 1 *Comme $\sigma_{i+1} = \eta[y, \overrightarrow{cla}(y), \sigma_i]$ tel que $\neg y \in \sigma_i$, nous avons $\sigma_i \not\subset \overrightarrow{cla}(y)$. La prochaine étape de résolution ne peut pas impliquer les clauses contenant le littéral $\neg y$. Dans le cas contraire, le littéral y dans le graphe d'implications peut admettre deux explications possibles (implications), ce qui contredit la définition du graphe d'implications. Par conséquent, σ_i ne peut pas subsumer des clauses de $\mathcal{C}_{\sigma_k} - \mathcal{C}_{\sigma_i}$.*

Propriété 2 *Soit $\mathcal{F}$ une formule et π une dérivation de la clause assertive . Si $\sigma_i \in \pi$ subsume une clause c de $\mathcal{C}_{\sigma_i}$ alors c est $\mathcal{C}_{\sigma_i}$-subsumé modulo PU.*

Preuve 2 *Comme $\sigma_i \in \pi$ est générée à partir de $\mathcal{C}_{\sigma_i}$ par résolution, alors $\mathcal{C}_{\sigma_i} \models \sigma_i$. Par définition de la dérivation de la clause assertive et le graphe d'implications, on a aussi $\mathcal{C}_{\sigma_i} \models^* \sigma_i$ (voir section 2.3). Comme σ_i subsumes c ($\sigma_i \subseteq c$), nous déduisons que $\mathcal{C}_{\sigma_i} \models^* c$.*

La propriété 2 montre que si une clause c codée dans le graphe d'implications est subsumée par σ_i, alors une telle subsumption peut être capturée par la subsumption modulo PU, alors que la propriété 1 mentionne que le test de vérification de la subsumption de σ_i peut être restreint aux clauses de $\mathcal{C}_{\sigma_i}$. Par conséquent, une première approche de subsumption dynamique des clauses peut être définit comme suit : Soit $\pi = \langle \sigma_1, \ldots, \sigma_i, \ldots, \sigma_k \rangle$ une dérivation de la clause assertive . Pour chaque résolvante $\sigma_i \in \pi$, nous appliquons le test de vérification de la subsumption entre σ_i et l'ensemble de clauses $\mathcal{C}_{\sigma_i}$,

Dans la suite, nous montrons que nous pouvons réduire encore plus l'ensemble des clauses dans les tests de subsumption en considérant seulement un sous-ensemble de $\mathcal{C}_{\sigma_i}$. Évidemment, comme σ_i est une résolvante de la dérivation de la clause assertive π, alors il existe deux chemins à partir reliant les nœuds conflictuels x et $\neg x$ respectivement aux nœuds associés aux littéraux de σ_i affectés au niveau du conflit. Par conséquent, on a la propriété suivante.

Propriété 3 *Soit π une dérivation de la clause assertive , $\sigma_i \in \pi$ et $c \in \mathcal{C}_{\sigma_i}$. Si σ_i subsumes c, alors il existe deux chemins liant les nœuds conflictuels x et $\neg x$ respectivement, à un ou plusieurs nœuds associés aux littéraux de la clause σ_i affectés au niveau du conflit.*

La preuve de la propriété est immédiate du moment que $\sigma_i \subset c$. Comme cette propriété est vraie pour σ_i qui est générée par résolution à partir des deux clauses contenant

x et $\neg x$, alors elle reste vraie aussi pour son super-ensemble (c).

Pour une σ_i donnée, la propriété 3 mène à une autre restriction de l'ensemble des clauses à considérer pour les tests de subsumption. Nous considérons que l'ensemble des clauses $\mathcal{P}_{\sigma_i}$, liées (par des chemins) aux nœuds conflictuels x et $\neg x$.

Illustrons cette restriction en utilisant l'exemple 1 (voir aussi la Figure 1). Soit $\pi = \langle \sigma_1, \ldots, \sigma_7 \rangle$ tel que $\sigma_7 = (\neg x_{11} \vee \neg x_{12} \vee \neg x_6 \vee \neg x_1)$. On a $\mathcal{C}_{\sigma_7} = \{c_1, c_2, c_3, c_4, c_5, c_6, c_7, c_8\}$ et $\mathcal{P}_{\sigma_7} = \{c_1, c_2, c_4, c_5, c_6, c_8\}$. En effet, à partir du des nœuds de la clause c_3 on a seulement qu'un seul chemin vers le nœud x_9. par conséquent, la clause c_3 peut être enlevée de l'ensemble des clauses à vérifier par subsumption. De façon similaire, partant de la clause c_7 on ne peut atteindre qu'un seul nœud conflictuel en l'occurence x_9 à partir de x_7. Donc, c_7 peut être omise de l'ensemble des clauses susceptibles d'être subsumées.

Propriété 4 *Soit $\pi = \langle \sigma_1, \ldots, \sigma_k \rangle$ une dérivation de la clause assertive . La complexité temporelle de notre approche de subsumption dynamique est en $O(|\mathcal{C}_{\sigma_k}|^2)$.*

Preuve 3 *À partir de la définition de $\mathcal{C}_{\sigma_i}$, nous avons $|\mathcal{C}_{\sigma_i}| = i + 1$. Dans le pire des cas, on doit considérer $i + 1$ tests de subsumption. Donc pour toute σ_i avec $1 \leq i \leq k$, on doit tester $\sum_{1 \leq i \leq k}(i + 1) = \frac{k \times (k+3)}{2}$. Comme $k = |\mathcal{C}_{\sigma_k}|$, la complexité dans le pire des cas est en $O(|\mathcal{C}_{\sigma_k}|^2)$.*

La complexité dans le pire des cas est quadratique si on considère $\mathcal{P}_{\sigma_k} \subset \mathcal{C}_{\sigma_k}$.

3.3 Subsumption dynamique

Dans la section 3.2, nous avons présenté l'approche générale de subsumption dynamique. Sa complexité est quadratique en le nombre de clauses utilisées dans la dérivation de la clause assertive. Comme indiqué dans l'introduction, pour cencevoir une technique de subsumption dynamique et efficace, nous avons besoin de trouver un compromis entre le temps de calcul et la qualité de la simplification. Dans cette section, on propose une restriction du schéma général de subsumption dynamique présenté, appelé subsumption dynamique à la volée. Celle-ci applique la subsumption seulement entre la résolvante courante σ_i et la dernière clause du graphe d'implications utilisée pour sa dérivation. Plus précisemment, supposons que $\sigma_i = \eta[y, c, \sigma_{i-1}]$, nous testons seulement la subsumption entre σ_i et c.

La propriété suivante montre une condition suffistante permettant d'éliminer y de c.

Propriété 5 *Soit π une dérivation de la clause assertive, $\sigma_i \in \pi$ telle que $\sigma_i = \eta[y, c, \sigma_{i-1}]$. Si $\sigma_{i-1} - \{y\} \subseteq c$, alors c est subsumée par σ_i.*

Preuve 4 *Soit $c = (\neg y \vee \alpha)$ et $\sigma_{i-1} = (y \vee \beta)$. Alors $\sigma_i = (\alpha \vee \beta)$. Comme $\sigma_{i-1} - \{y\} \subseteq c$, donc $\beta \subseteq \alpha$. D'où, $\sigma_i = \alpha$ qui subsume $(\neg y \vee \alpha) = c$.*

En considérant les solveurs SAT modernes inculant l'analyse de conflits, l'intégration de l'approche de subsumption dynamique peut être faite avec un coût négligeable. En effet, en utilisant un simple compteur, durant la procédure de l'analyse de conflits, on peut vérifier la condition suffisante donnée dans la propriété 5 en temps constant. En effet, à chaque étape du processus de dérivation de la clause assertive, on génére la prochaine résolvante σ_i à partir de la clause c et l'ancienne résolvante σ_{i-1}. Dans l'implémentation classique de l'analyse de conflit, on peut vérifier en temps constant si un littéral donnée est présent dans la résolvante courante. Par conséquent, durant le parcours de la clause c, on calcule additionnellement le nombre n des littéraux de c qui appartiennent à σ_{i-1}. Si $n \geq |\sigma_{i-1}| - 1$ nous pouvons conclure que c est subsumée par $\sigma_i = \eta[y, c, \sigma_{i-1}]$.

4 Expérimentations

Les expérimentations menées sont effectuées sur un large panel d'instances industrielles et crafted. Toutes les instances sont simplifiées par pré-traitement `SatElite` [8]. Nous avons intégré notre approche de subsumption dynamique dans `Minisat` [9] et `Rsat` [15] et nous avons effectué une comparaison entre les résultats obtenues par les solveurs originaux et ceux incluant l'approche de subsumption dynamique. Tous les tests sont effecutées sur un cluster Xeon 3.2GHz (2 GB RAM). Les Résultats du temps de calcul sont indiqués en secondes.

4.1 Problèmes crafted

Pour ces expérimentations, le temps de calcul limite est fixé à 3 heures. Ces problèmes sont connues pour être difficile pour tous les solveurs DPLL existants. Ils contiennent par exemple les instances "Quasi-group", instances SAT forcées aléatoires, comptage, problèmes sociaux du golfeur, etc.

Le schéma (en échelle logarithmique) donnée dans la partie haute de la Figure 2 détaille les résultats de `Minisat` et `Minisat+SD` sur chaque instance crafted. L'axe x (resp. y) corresponds au temps CPU tx (resp. ty) obtenu par `Minisat` (resp. `Minisat+SD`). chaque point de coordonnées (tx, ty), corresponds à une instance SAT. Les points en-dessous (resp. au-dessus) de la diagonale indiques les instances résolues plus rapidement en utilisant

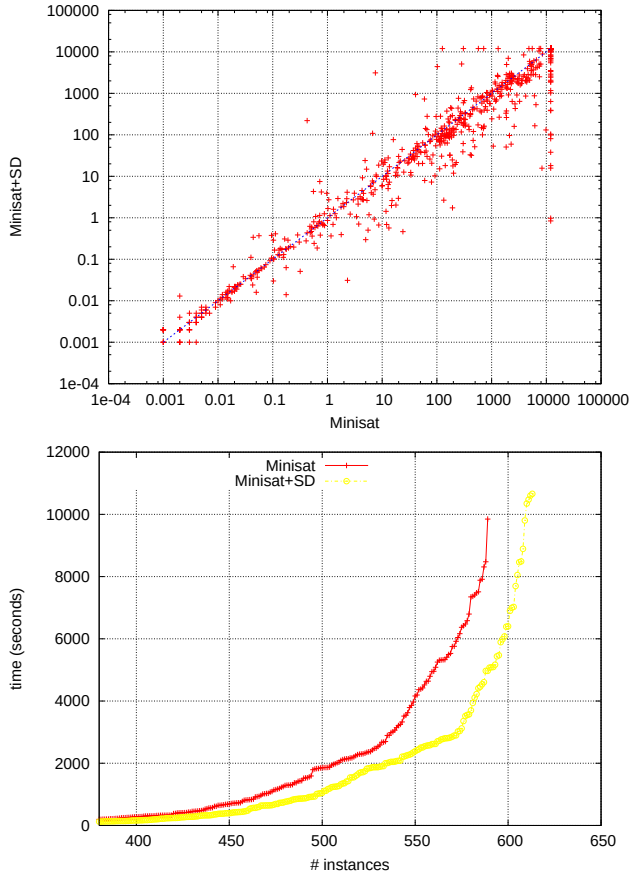


FIG. 2 – Problèmes crafted : `Minisat` vs `Minisat+SD`

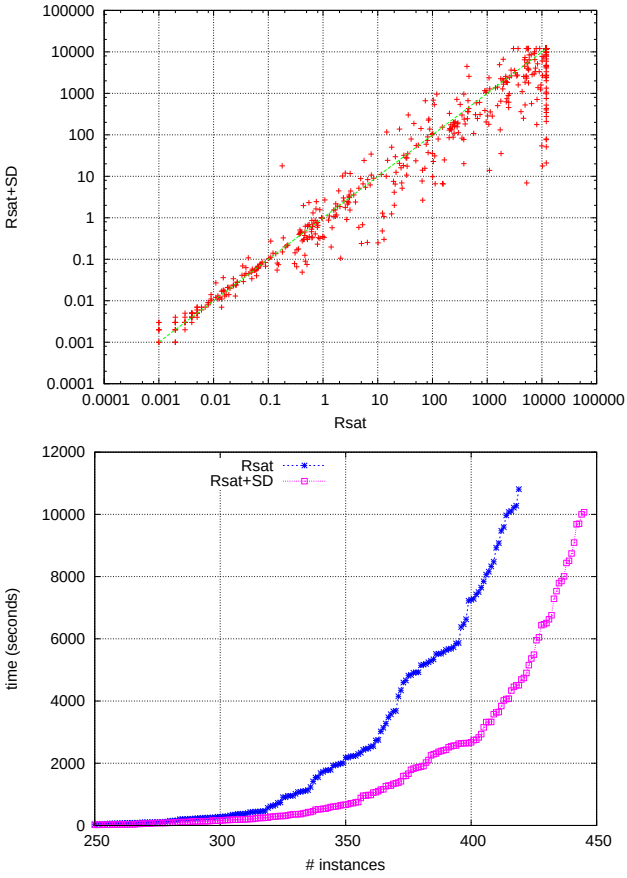


FIG. 3 – Problèmes crafted : `Rsat` vs `Rsat+SD`

la subsumption dynamique i.e., $ty < tx$ (resp. $ty > tx$). Cette figure montre clairement le gain de temps obtenu grâce à l'approche de subsumption dynamique. En nombre de points, nous avons trouvé que 365 instances sont résolues efficacement en intégrant la subsumption dynamique. Dans certains cas, nous avons des gains atteignant un ordre de magnitude. Bien évidemment, il existe des instances où la subsumption dynamque décroît les performances de `Minisat` (178 instances).

La partie en bas de la figure 2 montre les mêmes résultats avec une représentation différente donnant pour chaque technique le nombre d'instances résolues (# instances) en moins de t secondes. Cette figure confirme l'efficacité de l'approche présentée sur ces problèmes. Sur plusieurs classes, le nombre de clauses subsumées est important e.g., `x_*`, `QG_*`, `php_*`, `parity_*`. Sur les `genurq_*`, `mod_*`, et `urquhart_*` les problèmes sont simplifiés chaque conflit sur 4.

La Figure 3 montre les résultats de `Rsat` et `Rsat+SD`. Globalement, on peut voir que l'ajout de la subsumption dynamique à `Rsat` améliore ces performances. Une analyse fine de la partie gauche de la figure 3 montre que `Rsat+SD` résoud plus rapidement 327 instances que `Rsat`, qui lui même résoud 219 problèmes plus efficacement que son son opposé.

À notée que les performances de `Rsat` et `Rsat+SD` sont inférieures à celles de `Minisat` et `Minisat+SD`. Ceci est dû principalement au faite que `Rsat` et `Rsat+SD` utilise une politique de redémarrage plus rapide qui reste moins efficace sur les instances de type crafted.

4.2 Problèmes industriels

Comme pour les instances crafted, sur cette catégorie de problèmes, le temps limite est fixé à 3 heures aussi. La table 1 détaille les résultats sur les problèmes SAT industriels issus des deux compétitions SAT'2007 et SatRace'2008. La première colonne représente les familles d'instances. La seconde (#inst.) indique le nombre total des instances contenu dans chaque famille. Les colonnes suivantes indiquent les résulats respectives pour `Rsat`, `Rsat+SD`, `Minisat` et `Minisat+SD`. Dans chacune de ces colonnes, le premier nombre représente le nombre d'instances résolues et le nombre entre parenthèses indique le nombre d'instances résolues plus rapidement que son opposé. La dernière ligne de la table indique le nombre total de chaque colonne. On peut voir que `Rsat+SD` et `Minisat+SD` sont généralement plus rapides et résolvent plus de problèmes que `Rsat` et `Minisat` respectivement.

La table 2, représente un focus sur quelques familles industrielles. Les améliorations sont relativement importants. Si on considère la famille vmpc, on peut voir que notre simplication dynamique permet d'avoir un gain d'un ordre de magnitude avec `Rsat+SD` (instances 24 et 25). Sur la même famille, sur les 9 instances résolues par `Rsat+SD` et `Rsat`, `Rsat+SD` est meilleure 8 fois. De sont côté `Minisat+SD` est meilleur 5 fois sur 7.

Globalement, nos expérimentations nous ont permis de démontrer deux choses. Premièrement, notre technique ne dégrade pas mais au contraite améliore souvent les performances des solveurs DPLL sur les problèmes indistriels. Second, Elle améliore l'applicabilité de ces algorithmes sur des classes de problèmes qui sont faites pour être difficiles pour eux. Vu que l'implémentation de notre algorithme est aussi simple, nous pensons que de manière globale, elle resprésente une contribution intéréssante dans une recherche de robustesse des solveurs DPLL modernes.

5 Travaux précédents

Dans Darras et al. [5], les auteurs proposent un prétraitement basé sur la propagation unitaire pour la déduction des sous-clauses. En considérant le graphe d'implications généré par le processus de la propagation de contraintes comme un arbre de résolution, l'approche proposée déduit des sous-clauses à partir des clauses originales de la formule. Cependant, l'approche dynamique proposée par les auteurs est coûteuse en temps. L'évaluation expérimentale est donnée seulement en terme de nombre de nœuds.

Dans [17], un algorithme pour maintenir la base de clauses fermée par subsumption est présentée. Elle détecte efficacement et applique la subsumption lorsqu'une clause est ajoutée. En plus, l'algorithme compacte avidement la base des clauses en appliquant récursivement la résolution dans le but de réduire la taille de la base.

"Minimisation de la clause conflit" est introduite dans [14]. Elle est aussi implémentée dans Minisat 1.14 [7] et PicoSAT [4]. Elle élimine les littéraux des clauses apprises en effectuant la résolution de façon récursive avec les clauses du graphe d'implications. Remarquons que dans les expérimentations effectuées les solveurs de base utilisées `Minisat` et `Rsat` implémentent déjà cette technique.

6 Conclusion

Ce travail présente une nouvelle technique de subsumption des formules booléennes sous forme CNF. Il utilise de façon originale l'apprentissage pour réduire les clauses originales ou apprises. À chaque conflit et durant le processus de dérivation de la clause assertive, la subsumption entre les résolvantes intermédiaires générées et des clauses codées dans le graphe d'implications est testée dans le but de les simplifier en vérifiant une condition suffisante. Il est a noter que comme notre technique de subsumption repose sur les clauses utilisées dans la dérivation de la clause assertive, elle tend à simplifier la partie de formule identifiée

familles	# inst.	Rsat	Rsat+SD	Minisat	Minisat+SD
IBM_*	53	15(7)	**17**(**10**)	15(**10**)	15(7)
APro_*	16	12(**7**)	12(5)	**14**(6)	13(**8**)
mizh_*	10	10(**7**)	10(3)	10(5)	10(5)
Partial_*	20	6(2)	**7**(**5**)	1(0)	**2**(**2**)
total_*	20	13(6)	13(**8**)	**10**(5)	9(**6**)
dated_*	20	15(6)	**16**(**10**)	**14**(**10**)	13(4)
braun_*	7	4(1)	4(**3**)	5(2)	5(**3**)
velev_*	10	2 (0)	2(**2**)	**2**(**2**)	1(0)
sort_*	5	2(**2**)	2(0)	2(0)	2(**2**)
manol_*	10	8(3)	8(**5**)	8(**5**)	**9**(4)
vmpc_*	10	9(1)	9(**8**)	6(2)	**7**(**5**)
clause_*	5	3(**2**)	3(1)	3(0)	3(**3**)
cube_*	4	4(2)	4(2)	4(1)	4(**3**)
gold_*	4	2 (2)	2 (**0**)	2(0)	2(**2**)
safe_*	4	2(0)	2(**2**)	1(0)	1(**1**)
simon_*	5	5 (**4**)	5 (1)	5(**3**)	5(2)
block_*	2	2(**2**)	2(0)	2(0)	2(**2**)
dspam_*	10	10(5)	10(5)	10(5)	10(5)
schup_*	3	3(2)	3(1)	3(**2**)	3(1)
post_*	10	8(3)	8(**5**)	5(3)	**6**(3)
ibm_*	20	20(6)	20(**14**)	19(6)	19(**13**)
Total	248	155(70)	**159**(**90**)	141(67)	141(**81**)

TAB. 1 – Problèmes industriels

comme étant importante par la stratégie de recherche basée sur l'activité.

Les résultats expérimentaux montrent que l'intégration de notre méthode au sein de deux solveurs `Minisat` et `Rsat` issus de l'état de l'art de SAT bénéficie particulierement aux problèmes crafted et permet des améliorations significatives sur plusieurs familles industrielles.

Comme travaux futures, nous envisageons d'étendre notre approche en visant une subsumption plus exhaustive des clauses. Une autre voie de recherche consiste à exploiter ce cadre de subsumption pour affiner l'heuristique de choix de variable. En effet, à chaque fois qu'un littéral est éliminé, celà signifie qu'on s'est trompé de clause conflit et que le graphe d'implication doit changer en conséquence et du même coup l'ensemble des variables à pondérer par l'heuristique.

Références

[1] Roberto J. Bayardo, Jr. and Robert C. Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI'97)*, pages 203–208, 1997.

[2] Paul Beame, Henry A. Kautz, and Ashish Sabharwal. Understanding the power of clause learning. In Georg Gottlob and Toby Walsh, editors, *IJCAI*, pages 1194–1201. Morgan Kaufmann, 2003.

[3] A Biere and N. Eén. Effective preprocessing in sat through variable and clause elimination. In *Proceedings of the Eighth International Conference on Theory and Applications of Satisfiability Testing (SAT'05)*, 2005.

[4] Armin Biere. Picosat essentials. *Journal on Satisfiability, Boolean Modeling and Computation (JSAT)*, 4(1) :75–97, 2008.

[5] S. Darras, G. Dequen, L. Devendeville, B. Mazure, R. Ostrowski, and L. Saïs. Using Boolean constraint propagation for sub-clauses deduction. In *Proceedings of the Eleventh International Conference on Principles and Practice of Constraint Programming(CP'05)*, pages 757–761, 2005.

[6] M. Davis, G. Logemann, and D. W. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7) :394–397, 1962.

[7] N. Een and N. Sörensson. Minisat - a sat solver with conflict-clause minimization. In *Proceedings of the Eighth International Conference on Theory and Applications of Satisfiability Testing (SAT'05)*, 2005.

[8] N. Eén and A. Biere. Effective preprocessing in SAT through variable and clause elimination. In *Proceedings of the Eighth International Conference on Theory and Applications of Satisfiability Testing (SAT'05)*, pages 61–75, 2005.

[9] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In *Proceedings of the Sixth International*

instances	Rsat	Rsat+SD	Minisat	Minisat+SD
vmpc_33	5540	**1562**	–	–
vmpc_29	2598	**1302**	–	**1252**
vmpc_30	366	**105**	3111	**2039**
vmpc_27	593	**327**	1159	**637**
vmpc_31	–	–	–	–
vmpc_25	39	**1**	830	**318**
vmpc_26	182	**69**	1239	**1235**
vmpc_34	3366	**944**	–	–
vmpc_24	43	**8**	**82**	210
vmpc_28	**173**	488	**3859**	5478
partial-5-11-s	931	**176**	–	**2498**
partial-5-13-s	503	**71**	3248.38	**669**
partial-5-15-s	**737.064**	825	–	–
partial-10-11-s	1242	**875**	–	–
partial-5-19-s	1134	**498**	–	–
partial-5-17-s	**7437.82**	10610	–	–
partial-10-13-s	–	**3237**	–	–
ibm-2002-04r-k80	90	**33**	**113**	152
ibm-2002-11r1-k45	67	**29**	102	**65**
ibm-2002-18r-k90	265	**157**	1044	**769**
ibm-2002-20r-k75	**36**	185	2112	**668**
ibm-2002-22r-k60	738	**691**	5480	**3434**
ibm-2002-22r-k75	363	**349**	1109	**688**
ibm-2002-22r-k80	**285**	298	894	**642**
ibm-2002-23r-k90	1477	**965**	7127	**2670**
ibm-2002-24r3-k100	273	**256**	**133**	249
ibm-2002-25r-k10	**3104**	3118	**2877**	3172
ibm-2002-29r-k75	353	**248**	**272**	1107
ibm-2002-30r-k85	3853	**592**	–	–
ibm-2002-31_1r3-k30	1203	**652**	1150	**998**
ibm-2004-01-k90	114	**30**	**251**	726
ibm-2004-1_11-k80	394	**222**	559	**329**
ibm-2004-23-k100	**326**	687	3444	**2743**
ibm-2004-23-k80	**465**	563	2060	**1584**
ibm-2004-29-k25	290	**210**	1061	**1017**
ibm-2004-29-k55	533	**16**	558	**124**
ibm-2004-3_02_3-k95	**1**	2	**1**	2

TAB. 2 – Zoom sur quelques familles industrielles

Conference on Theory and Applications of Satisfiability Testing (SAT'03), pages 502–518, 2002.

[10] Carla P. Gomes, Bart Selman, and Henry Kautz. Boosting combinatorial search through randomization. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI'97)*, pages 431–437, Madison, Wisconsin, 1998.

[11] H. Kautz, E. Horvitz, Y. Ruan, C. Gomes, and B. Selman. Dynamic restart policies. In *Proceedings of the Eighteenth National Conference on Artificial Intelligence (AAAI'02)*, pages 674–682, 2002.

[12] Joao P. Marques-Silva and Karem A. Sakallah. GRASP - A New Search Algorithm for Satisfiability. In *Proceedings of IEEE/ACM International Conference on Computer-Aided Design*, pages 220–227, November 1996.

[13] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff : Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference (DAC'01)*, pages 530–535, 2001.

[14] Alexander Nadel. Backtrack search algorithms for propositional logic satisfiability : Review and innovations. Master's thesis, Master Thesis, the Hebrew University, November 2002.

[15] Knot Pipatsrisawat and Adnan Darwiche. A lightweight component caching scheme for satisfiability solvers. In *Proceedings of 10th International Conference on Theory and Applications of Satisfiability Testing(SAT)*, pages 294–299, 2007.

[16] Sathiamoorthy Subbarayan and Dhiraj K. Pradhan. NiVER : Non-increasing variable elimination resolution for preprocessing SAT instances. In *Proceedings of the Seventh International Conference on Theory and Applications of Satisfiability Testing (SAT'04)*, pages 276–291, 2004.

[17] Lintao Zhang. On subsumption removal and on-the-fly cnf simplification. In *SAT'2005*, pages 482–489, 2005.

Des contraintes globales prêtes à brancher

Guillaume Richaud **Xavier Lorca** **Narendra Jussien**

École des Mines de Nantes, LINA UMR CNRS 6241, F–44307 Nantes Cedex 3, France
{guillaume.richaud,xavier.lorca,narendra.jussien}@emn.fr

Résumé

Les contraintes globales représentent un outil indispensable dans la résolution et la modélisation de problèmes combinatoires. Il s'agit d'un concept clé de la programmation par contraintes qui, par sa généralité, peut s'étendre à d'autres paradigmes. Pourtant, concevoir, implémenter et maintenir des contraintes globales sont des exercices difficiles, très liés au solveur sous-jacent dans lequel elles sont implémentées. Cet article vise à jeter les bases de contraintes *prêtes à brancher* clairement découplées du solveur hôte. Ainsi, trois points sont particulièrement étudiés : comment une contrainte globale est liée à un solveur ? comment rendre indépendantes du solveur les structures de données internes des contraintes ? dans quelle mesure ce découplage introduit-il un surcoût ?

Abstract

Global constraints represent invaluable modeling tools for Constraint Programming (CP). They represent more than ever a key feature for the technology and even more generally for all search paradigms. However, designing, implementing and maintaining global constraints is a tedious task, from a practical point of view, generally strongly related to the solver. In this context, this paper aims at paving the way for pluggable constraints that allows a clear separation of the global constraints from the host solver. For this purpose, three bolts are studied : (1) How a global constraint is connected with its host solver ? (2) How the internal data structures of a global constraint could be made independent from the related solver ? (3) Is there a real overhead due to this decoupling ?

1 Introduction

La programmation par contraintes est considérée maintenant comme une technique de choix pour traiter de problèmes combinatoires complexes. Par contre, la quête du Graal décrite par Eugene Freuder il y a quelques années n'est toujours pas achevée : résoudre un problème avec la programmation par contraintes n'est toujours pas à la portée d'un seul clic de souris. Choisir le bon solveur, l'intégrer à l'applicatif existant, modéliser proprement et efficacement le problème, trouver la bonne heuristique de recherche restent des étapes à la fois nécessaires et réservées à des experts du domaine. Récemment, certaines de ces étapes sont traitées par l'initiative de Jacob Feldman : CP inside[1]. Cette initiative a pour but de rendre la technologie contraintes disponible aux utilisateurs finaux par l'intermédiaire d'interfaces communément utilisées dans des outils commerciaux ou non (comme Microsoft Excel, Google apps, etc.). La possibilité d'embarquer la technologie dans des outils grand-public permettra d'introduire les moteurs décisionnels basés sur les contraintes comme des composants naturels des applicatifs métiers traditionnels.

Dans cet article, nous voudrions traiter un point important et orthogonal : la capacité de *brancher* une contrainte globale sur différents solveurs. Jusqu'à présent, les contraintes globales restent un composant-clé des solveurs de contraintes. Elles permettent de traiter efficacement de nombreux problèmes difficiles grâce à leur capacité à encapsuler des sous-problèmes récurrents dans un algorithme de résolution à base de propagation/filtrage. Elle représentent un des éléments phare de la programmation par contraintes et plus généralement des techniques de résolution (y compris les stratégies de recherche locale - hybrides, les solveurs à base d'explications/nogoods, etc.). Mais, développer une nouvelle contrainte globale est une tâche ardue et souvent ingrate. En effet, l'information généralement manipulée en provenance des variables (via leur domaine) est généralement trop légère pour développer un algorithme de filtrage efficace. Ainsi, les contraintes globales maintiennent généralement une structure de données interne complexe (on peut se référer au cas des contraintes sur les graphes pour s'en convaincre) pour implémenter des algorithmes de filtrage efficaces. Ainsi, la complexité d'im-

[1] http://4c.ucc.ie/web/posters/CPInsidePoster.pdf

plémentation d'une contraintes globale repose généralement sur le maintien efficace de cette structure de données interne [5, 2]. Aujourd'hui, les contraintes globales sont liées à un solveur de contraintes donné car elles reposent généralement sur les structures backtrackables de ce solveur. Ces structures sont utilisées pour le maintien efficace de propriétés utilisées par les algorithmes de filtrage (les composantes connexes d'un graphe par exemple) quand le solveur doit revenir en arrière. Ces structures sont généralement fortes consommatrices de CPU (avec des approches à base de recalcul) ou de mémoire (pour les approches à base de trailing ou de recopie) et ne permettent ainsi pas d'accéder aux dernières nouveautés du langage support (par exemple, dans le cas de Java, les *generics*). Ainsi, l'intérêt même de développer une nouvelle contrainte globale peut être remis en question car reposant sur des technologies dépassées.

L'objectif de cet article est de défricher les bases de ce que pourrait être un cadre général pour une séparation claire des contraintes globales de leur solveur hôte. Au delà même, il s'agit aussi de permettre l'utilisation des contraintes globales dans des paradigmes de recherche plus génériques : solveur à base d'explications [10] ou techniques de recherche locale [21]. Il ne s'agit, en aucun cas, d'un nouveau cadre de description des contraintes globales à l'image de ce que sont, par exemple, les propriétés de graphes [3] ou les contraintes *Range* et *Roots* [7], mais bien d'un schéma d'implémentation pour découpler une contrainte globale du solveur dans lequel elle a été développée (voir la figure 1). Nous nous intéressons à trois points particuliers :

- comment une contrainte globale est effectivement reliée au solveur sous-jacent ?
- comment les structures de données internes d'une contrainte globale peuvent être rendues indépendantes du solveur sous-jacent ?
- est-ce que le découplage introduit un sur-coût ?

Un premier pas significatif dans la fourniture d'un tel cadre général consiste à fournir un squelette général pour une contrainte indépendante de tout solveur (voir la figure 1). Ainsi, une contrainte *prête à brancher* peut être définie à l'aide de deux sortes d'objets : un type de donnée abstrait (TDA) qui décrit la structure de données interne et des algorithmes de filtrage. Les événements liés aux variables du solveur devant être traités par la contrainte, ils doivent en être *compris* pour qu'elle puisse mettre à jour le TDA et, de manière symétrique, les mises à jour du TDA doivent être répercutées du côté solveur. Ceci implique que ces mises à jours soient elles-même converties en événements sur les variables. C'est le rôle que joue l'interface présentée dans la figure 1.

Par la suite, nous commençons par décrire le contexte général de notre travail (section 3). Puis, nous discutons les différents points clés qui permettent de découpler proprement une contrainte de son solveur hôte (section 4). Après une introduction générale à l'approche (section 4.1), ce schéma de développement est appliqué à deux contraintes globales : la contrainte `boundAllDiff` [13] (section 4.2) et la contrainte `tree` [1] (section 4.3) ; et ce, dans le contexte de deux solveurs : `Choco` (`choco.emn.fr`) et `Gecode` (`gecode.org`). Alors, nous présentons une évaluation de l'utilisation pratique de ce schéma (section 5) en commençant par la comparison de chacune des contraintes *classiques* (`boundAllDiff` et `tree`) avec sa version prête à brancher (section 5.1). Ensuite, nous présentons un exemple pratique mêlant ces deux contraintes : le problème de chemin hamiltonien (section 5.2). Enfin, nous terminons par une discussion générale sur l'avenir de ces contraintes *prêtes à brancher*.

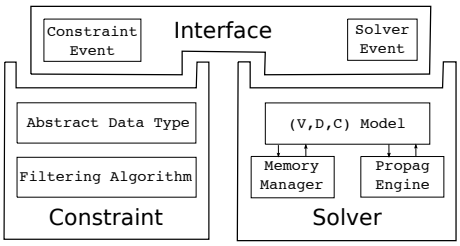


FIG. 1 – Une contrainte globale idéalement découplée de son solveur hôte.

2 Préliminaires

La programmation par contraintes (PPC) [12, 14, 20] est un domaine de recherche dont l'objectif est de fournir des outils logiciels déclaratifs et flexibles pour résoudre des problèmes combinatoires. Un problème de satisfaction de contraintes (CSP) est défini par un ensemble $\mathcal{V} = \{v_1, \ldots, v_n\}$ de variables (au sens mathématique du terme), un ensemble $\mathcal{D} = \{dom(v_1), \ldots, dom(v_n)\}$ de domaines qui représentent l'ensemble fini des valeurs possibles pour chacune des variables et un ensemble $\mathcal{C}$ de contraintes (relation logique sur un sous-ensemble des variables). Une solution pour un CSP est une affectation des variables (une valeur pour chaque variable) qui vérifie simultanément l'ensemble des contraintes du problème.

L'idée centrale de la programmation par contraintes est basée sur une technique de propagation-recherche qui consiste à explorer l'ensemble des affectations possibles pour les variables d'un CSP. Évidemment, cet espace de recherche est de taille exponentielle par nature. Ainsi, l'objectif est de détecter en amont (et donc de retirer) des portions qu'il est inutile d'explorer (car ne contenant aucune solution) de cet espace (sans évidemment explorer effectivement ces portions). Ceci est réalisé en utilisant des techniques de *filtrage* (identification et retrait de valeurs *inconsistantes* des domaines des variables en tenant compte des

contraintes du problème).

Dans ce contexte, les *contraintes globales* [6] représentent un outil indispensable de modélisation et de résolution dans le paradigme. En effet, une contrainte globale est un ensemble compact d'algorithmes et de solutions pour des sous-problèmes récurrents dans les CSP. Elles sont considérées comme des contraintes comme les autres mais généralement encapsulent un ensemble de contraintes définies sur un ensemble conséquent de variables.

Les contraintes globales offrent une vue concise et efficace du sous-problème par lequel elles sont définies. Elles utilisent de manière active la structure sous-jacente de ce sous-problème et proposent ainsi des algorithmes de filtrage particulièrement efficaces. En effet, la connaissance explicite embarquée dans cette structure permet une meilleure propagation (et donc une meilleure recherche de solution) en évitant de redécouvrir continuellement les mêmes incohérences (un tel phénomène est généralement appelé *thrashing*). Habituellement, les algorithmes mis en œuvre ont une complexité élevée mais ce surcoût est largement compensé par le pouvoir de filtrage apporté par la contrainte.

3 État de l'art

La conception, l'implémentation et la maintenance d'une contrainte globale est une tâche complexe d'un point de vue pratique. Aujourd'hui, des outils ou des concepts sont proposés pour faciliter le processus. Ainsi en est-il de la dérivation automatique d'algorithmes de filtrage pour des contraintes globales, des tentatives d'unification des architectures de solveurs ou encore des boîtes à outils pour l'implémentation de contraintes globales (ici dans le cadre des contraintes sur les graphes). Nous détaillons ces trois propositions.

La première proposition part du principe que l'implémentation d'une contrainte (globale) devrait être limitée en utilisant des propriétés de reformulation. Ainsi, les automates permettent d'exprimer la plupart des contraintes globales [3] et de produire automatiquement des vérificateurs de contraintes et des algorithmes de filtrage pour les contraintes [4] grâce à trois (méta) contraintes globales (`regular`, `cost regular` et `multi-cost regular`). Mais, toutes les contraintes ne peuvent pas être représentées par des automates et même si une telle représentation existe, le niveau de consistance atteint pour la génération automatique est fréquemment significativement plus faible qu'une approche dédiée. Une autre limitation de l'utilisation des automates est leur forte consommation mémoire. En effet, la plupart du temps, la taille de l'automate utilisé pour dériver un algorithme de filtrage n'est pas polynomial.

La seconde proposition est liée à l'architecture des solveurs. En deux mots, les solveurs de contraintes traitent la propagation selon l'une des deux philosophies suivantes : centrée sur les événements (comme dans choco) qui sont alors au cœur des files de propagation (la variable - et ce qui lui arrive - est alors au centre du système) ou centrée sur la propagation (comme dans gecode) où c'est plutôt la contrainte (ou plus précisément ses propagateurs - algorithmes directionnels de filtrage) qui est la plaque tournante du système. Une démarche intuitive introduite par [11] tente de concilier ces deux points de vue : une méthode générique (un *advisor*) permet d'interfacer de manière sûre les contraintes et les propagateurs qui lui sont liés. De cette façon, les *advisor*s représentent un moyen élégant de prendre en compte une propagation incrémentale dans les solveurs centrés sur la propagation tels que gecode. Ils peuvent être généralisés pour être considérés comme outils génériques de propagation dans les solveurs centrés sur les événements. Ainsi, une certaine souplesse est introduite dans le processus de propagation.

La troisième proposition est liée est la complexité de concevoir et d'implémenter des contraintes globales dans un domaine spécifique. Dans le cas des contraintes sur les graphes, un cadre générique a été proposé nommé `LS(Graph)`) par [8]. Ce cadre générique permet de s'affranchir de la complexité de la représentation des structures internes pour les contraintes sur les graphes. En effet, dans ce domaine, les structures de données sont clés pour obtenir des algorithmes efficaces surtout dans un contexte lié à la gestion du retour arrière. Plusieurs approches ont été proposées : une gestion *backtrackable* de la structure de données et le maintien incrémental d'une structure de données *ad hoc* [16]. On parle alors, dans ce dernier cas, d'une structure de données *pleinement dynamique*. C'est un premier pas vers une contrainte globale découplée de son solveur hôte.

Dans la suite de cet article, nous reprenons les deux dernières propositions afin de réaliser une séparation claire entre un solveur et ses contraintes. Plus précisément, nous montrons comment la notion d'*advisor* peut permettre d'interface proprement des contraintes *prêtes à brancher* avec différents solveurs. En effet, un développeur peut, dans ce contexte, spécifier précisément quand une contrainte doit être propagée selon l'observation des domaines des variables. Nous montrons aussi la complexité de rendre générique les structures de données internes des contraintes globales *prêtes à brancher* dans la gestion du retour arrière.

4 Les contraintes prêtes à brancher dans la pratique

Passer d'une contrainte dépendante du solveur à une contrainte prête à brancher nécessite de spécifier comment la contrainte considérée va interagir avec ledit solveur. Cela est généralement assez simple. En effet, en pratique, les algorithmes de résolution des contraintes gèrent les réveils

des contraintes selon les événements qui surviennent dans le domaine des variables sur lesquelles elles portent (dans le cadre des solveurs centrés événements), ou d'après des algorithmes généraux qui réalisent la propagation d'une manière donnée (pour les autres).

Dans le cadre des contraintes découplées des solveurs, les modifications dans les domaines des variables depuis le dernier réveil de la contrainte doivent être détectées et ensuite, traduites en événements gérables par la structure de données de la contrainte. De manière symétrique, nous avons à traduire les informations de filtrage de la contrainte en événements compréhensibles par le solveur (retraits de valeur, mise à jour des bornes, etc.) et applicables sur les domaines des variables qui peuvent être interprétées par l'algorithme de résolution basé sur les contraintes. Cette traduction bidirectionnelle des informations est appelée *interface* dans la suite. On peut remarquer qu'une interface peut être vue comme vision d'un niveau plus élevé des *advisors* introduits par [11], dans le sens où une interface s'occupe de diriger la stratégie des événements entre les contraintes et le solveur. Par conséquent, elle décide si une contrainte doit être propagée ou si le processus peut être retardé.

La transformation d'une contrainte globale en une contrainte prête à brancher devient complexe lorsqu'on considère des contraintes à états explicites. Pour résumer, une *contrainte à état explicite* gère sa propre structure de données afin de maintenir certaines propriétés qui sont utilisées par l'algorithme de filtrage pour retirer les valeurs inconsistantes dans les domaines des variables sur lesquelles la contrainte porte. Plus précisément, une contrainte globale s'appuie énormément sur les structures de données backtrackables du solveur hôte utilisé, et, dans le but de la rendre indépendante du solveur, nous devons gérer le retour arrière dans la contrainte explicitement. Par exemple, les contraintes arithmétiques binaires classiques ($\leq$, $=$, $\neq$) ne requièrent pas plus d'information que l'état du domaine de la variable. A l'opposé, les contraintes globales basées sur les graphes doivent modéliser une représentation du graphe dans le but de représenter efficacement et de manière expressive les propriétés liées (composantes connexes, composantes fortement connexes, nœuds dominants, arbres recouvrants, etc).

Dans la suite, nous présentons l'architecture d'un cadre d'interface pour implémenter des contraintes prêtes a brancher. Ensuite, nous illustrons une telle interface avec les contraintes `boundAllDiff` [13], qui est une contrainte globale simple (dans le sens qu'elle ne fait intervenir aucune structure de données interne), et `tree` [1], qui est une illustration typique d'une contrainte à état explicite (la structure de données interne représente un graphe orienté et des propriétés portant sur ce graphe). Finalement, nous branchons ces deux contraintes à deux solveurs différents.

4.1 Modèle des contraintes prêtes à brancher

Cette section étudie un modèle générique pour les contraintes prêtes a brancher. L'architecture est résumée par la figure 2. Elle est basée sur une décomposition en trois étapes, largement inspirée par le bien connu pattern *Observer* du génie logiciel orienté objet.

Une *Interface de Solveur* propose un ensemble de méthodes pour traduire les événements des variables du solveur en événements génériques gérables par l'interface dans la contrainte. Plus précisément, le solveur traduit les événements survenant sur les variables en événements génériques, et soumet ces événements au gestionnaire. Par exemple, supposons que le domaine d'une variable a changé, des valeurs ont été retirées (durant le filtrage ou l'énumération) ou ont été ajoutées (à cause d'un retour-arrière), alors un nouvel événement générique est généré et transmis au gestionnaire d'événements. Une *Interface de Contrainte* propose un ensemble de méthodes pour traduire les événements génériques fournis par le gestionnaire en événements compréhensibles par les structures de données internes de la contrainte. Les décisions de retrait (sur la structure interne) effectuées par les algorithmes de filtrage sont traduites de manière symétrique en événements génériques et sont renvoyés au gestionnaire. Un *Gestionnaire d'événements* qui distribue chaque événement générique. Il peut être vu comme une structure observable qui notifie les nouveaux éléments postés dans la queue d'événements génériques de l'interface de la contrainte prête à brancher correspondante ou de l'interface du solveur sous-jacent. Typiquement, si une variable du solveur a été instanciée, l'interface du solveur construit et poste un nouvel événement générique dans la queue du gestionnaire. Ensuite, le gestionnaire notifie l'interface de la contrainte que de nouveaux événements sont disponibles. De manière similaire, si l'algorithme de filtrage change la structure interne de la contrainte alors, un événement générique est produit et posté dans la queue du gestionnaire. L'interface du solveur est alors notifiée qu'un nouvel événement est disponible. Cependant, on peut remarquer que même si une traduction d'un événement du solveur en un événement contrainte est généralement direct, l'opération inverse peut être compliquée. Par exemple, dans le cadre des contraintes basées sur des graphes, les événements haut niveau du solveur peuvent être des retraits de valeurs dans les domaines des variables. De tels événements sont interprétés comme un ensemble de retraits d'arc dans la structure de données modélisant le graphe. De manière similaire, un retour-arrière consiste en un ajout d'un ensemble de valeurs dans les domaines des variables, et est interprété comme l'ajout d'un ensemble d'arcs dans la structure de données modélisant le graphe.

Dans la suite, nous ne détaillerons pas l'implémentation de notre interface générique mais nous détaillerons les différentes difficultés qui peuvent être rencontrées en fonc-

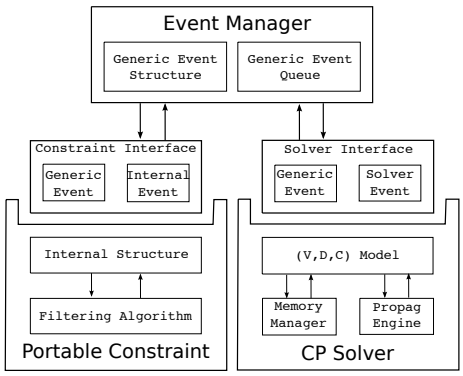


FIG. 2 – Interfaçage d'un solveur et d'une contrainte *prête à brancher* par l'intermédiaire du gestionnaire d'événements.

tion du type de contrainte et des algorithmes de résolution. Nos expérimentations sont basées sur les solveurs `Choco` et `Gecode`. Nous avons décidé volontairement de considérer deux types de contraintes globales : la contrainte `boundAllDiff` qui n'utilise aucune structure de données internes et la contrainte `tree` dont la structure de données interne est basée sur une représentation de graphe qui implique de véritables traductions des événements touchant les domaines en événements liés au graphe.

4.2 Une interface pour une contrainte globale simple : le cas `boundAllDiff`

Nous montrons d'abord, que dans le cas des contraintes globales qui ne requièrent pas de structures de données backtrackables (c.-à-d., toutes les inférences sur les domaines sont faites par un raisonnement direct sur les domaines des variables) alors, proposer une version prête à brancher de ces contraintes est facile. Basé sur l'algorithme de consistance de bornes introduit dans [13], cette section présente une implémentation prête à brancher de la contrainte `boundAllDiff`. Nous soulignons que la difficulté d'implémenter une telle contrainte prête à brancher est uniquement lié à l'interface avec le solveur.

Soit $v_1, v_2, \ldots, v_n$ un ensemble de variables avec les domaines finis $dom(v_1)$, $dom(v_2), \ldots,$ $dom(v_n)$, et des contraintes de différences deux à deux dont l'ensemble est noté `allDiff`$(v_1, \ldots, v_n)$ défini par $\{(e_1, \ldots, e_n) \mid \forall e_i \in dom(v_i), e_i \neq e_j \text{ for } i \neq j\}$. Il s'agit d'une contrainte très étudiée. Le premier algorithme de filtrage pour rendre les domaines consistants a été introduit par Régin dans [15]. Ici, nous rappelons l'algorithme de filtrage propageant la consistance aux bornes introduit dans [13]. Afin de fournir des approches de filtrage rapides et pouvant s'appliquer à des problèmes de grande taille, plusieurs algorithmes de filtrage (qui atteignent un niveau de consistance plus faible que la consistance de domaine) ont été introduits. Le plus rapide est l'algorithme de consistance aux bornes, `boundAllDiff`. Intuitivement, au plus n variables peuvent avoir leur domaine inclus dans un intervalle contenant n valeurs. Dans un intervalle de Hall (un intervalle I de taille n, tel qu'il y ait n variables dont le domaine est contenu dans I), une solution utilisera toutes les valeurs pour ces variables rendant ses valeurs inutilisables pour les autres variables. Ainsi, pour maintenir la consistance aux bornes, nous devons vérifier pour chaque intervalle I qu'il y a, au moins, autant de valeurs que de variables dont le domaine est inclus dans I. Et pour chaque intervalle de Hall I, nous devons interdire toutes les valeurs de I dans les autres variables.

Definition 1 (`boundAllDiff` – consistance) *Une contrainte `boundAllDiff`, portant sur les variables $(x_1, \ldots, x_n)$, est consistante aux bornes si et seulement si les conditions suivantes sont vérifiées : (1) $|D_i| \geq 1 (i = 1, \ldots, n)$, (2) pour chaque intervalle I : $|K_I| \leq |I|$, et (3) pour chaque intervalle de Hall I, $\{min(x_i), max(x_i)\} \cap I = \emptyset$ for all $x_i \notin K_I$.*

Pour chaque réveil, la contrainte `boundAllDiff` trie les variables et initialise plusieurs compteurs, ainsi aucune structure de données n'est maintenue entre deux appels de la procédure de filtrage. Les informations requises par la contrainte est l'ensemble des variables avec les valeurs maximum et minimum de chaque domaine. Alors, pour être capable de brancher la contrainte `boundAllDiff` dans différents solveurs, l'interface doit fournir une vue générique des variables. Ainsi, pour une contrainte `boundAllDiff` prête à brancher, le cadre générique nécessite l'implémentation de : **(1)** L'interface du solveur donne le minimum et le maximum du domaine de chaque variable et, envoie une vue de ces domaines au gestionnaire d'événements. Symétriquement, les événements de la contrainte envoyés par le gestionnaire d'événements sont traduits en événements de solveur et sont finalement convertis en modification de domaine des variables. **(2)** Le gestionnaire d'événements peut être vu ici comme une fonction bijective qui convertit les événements solveur en événements contrainte et vice-versa. **(3)** L'interface contrainte contient l'algorithme de filtrage `boundAllDiff`. Les événements contrainte générés par l'algorithme de filtrage consistent uniquement en un ensemble de modifications portant sur les valeurs minimale et maximale des vues des variables dans la structure de la contrainte.

La complexité de l'interface dépend des informations requises par la contrainte et des informations disponibles dans l'algorithme de recherche. Généralement, les valeurs maximale et minimale pour une variable sont accessibles en temps constant, aussi l'interface a une complexité de $O(n)$. De plus, la structure de données et l'algorithme de filtrage sont exactement les mêmes dans la version prête à brancher ou dans la version *ad hoc* de la contrainte.

4.3 Le cas des contraintes globales à états : la contrainte tree

Dans le cas de contraintes globales qui reposent sur une structure de données backtrackable (quelques inférences ont besoin de détecter de manière répétée un motif caché dans les domaines des variables) alors, proposer une version prête à brancher de ces contraintes est plus délicat, et des algorithmes *pleinement dynamiques* sont nécessaires pour fournir une contrainte indépendante du solveur. De plus, comme une contrainte prête à brancher n'est pas dépendante du solveur hôte, un choix plus large de structures de données est offert. Cette liberté permet de choisir et d'implémenter la plus efficace. Cette section illustre ce point avec la contrainte `tree` [1].

La contrainte `tree` est représentée par un graphe orienté $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ dans lequel les nœuds représentent les variables, $\mathcal{V} = \{v_1, \ldots, v_n\}$, les arcs représentent la relation de succession directe entre eux, $dom(v_i) = \{j \mid (v_i, v_j) \in \mathcal{E}\}$, et `ntree` est une variable qui spécifie le nombre d'arbres dans la forêt (<u>`ntree`</u> et $\overline{\texttt{ntree}}$ correspondent respectivement à la valeur minimal et à la valeur maximale de $dom(\texttt{ntree})$). Une contrainte $\texttt{tree}(\texttt{ntree}, \mathcal{G})$ spécifie que le graphe orienté associé $\mathcal{G}$ devrait être une forêt de `ntree` arbres, formellement :

Definition 2 *Une* ground instance *de la contrainte* $\texttt{tree}(\texttt{ntree}, \texttt{VER})$ *est une* solution *si et seulement si (1) le graphe orienté associé $\mathcal{G}$ est composé de* `ntree` *composantes connexes, et (2) chaque composante connexe de $\mathcal{G}$ n'a pas de circuit impliquant plus d'un nœud (notons que chaque composante contient exactement un nœud possédant une boucle et qui correspond à la racine de l'arbre).*

Étant donné un graphe orienté $\mathcal{G}$ et un ensemble de *de racines potentielles*[2], le nombre minimal d'arbres (<u>`ntree`</u>) pour partitionner le graphe orienté $\mathcal{G}$ associé à une contrainte `tree` est le nombre de puits (c.-à-d. de nœuds sans arc sortant exceptée la boucle sur lui-même) du graphe réduit[3] de $\mathcal{G}$, et le nombre maximum d'arbres ($\overline{\texttt{ntree}}$) pour partitionner le graphe orienté $\mathcal{G}$ est le nombre de racines potentielles.

Ensuite, nous détaillons l'algorithme de filtrage de la contrainte `tree`. Lorsque la variable `ntree` doit atteindre la valeur $\overline{\texttt{ntree}}$, l'algorithme force, pour chaque racine potentielle, l'arc boucle (qui représente le fait qu'il s'agit d'une racine potentielle). Dans le cas où `ntree` doit atteindre <u>`ntree`</u>, l'algorithme retire, pour chaque racine potentielle qui n'appartient pas à une composante fortement connexe puits de $\mathcal{G}$, la boucle sur lui-même. Finalement, pour n'importe quel `ntree` la règle principale de filtrage associée avec la contrainte est basée sur la détection des *noeuds dominants* du graphe[4]. Alors, l'algorithme de filtrage doit détecter tous les nœuds de j de $\mathcal{G}$ tel qu'il existe un nœud i pour lequel j domine toutes les racines potentielles de $\mathcal{G}$ par rapport à i. Les arcs infaisables dans $\mathcal{G}$ pour une contrainte `tree` sont les arcs sortant (j, k), où j est un nœud dominant, tel qu'il n'existe pas un chemin de i à une racine potentielle de $\mathcal{G}$ utilisant l'arc (j, k).

Figure 3 décrit le squelette de la contrainte `tree` telle qu'elle a été implémentée dans le solveur `choco`. Ici, les effets des événements survenant sur les variables du graphe consistent en plusieurs modifications de la structure du graphe. Par exemple, si un arc (i, j) est retiré de $\mathcal{G}$ (c.-à-d., $j \notin dom(v_i)$) alors, ce retrait peut : **(1)** diminuer le nombre de racines potentielles, si $i = j$. Cela mène à une mise à jour de $\overline{\texttt{ntree}}$. **(2)** augmenter le nombre de composantes puits. Cela mémé a une augmentation de <u>`ntree`</u>. **(3)** augmenter le nombre de composantes fortement connexes (cfc) dans $\mathcal{G}$. Cela mène à modifier le graphe réduit $\mathcal{G}_r$ associé avec $\mathcal{G}$. **(4)** créer un nouveau nœud dominant dans $\mathcal{G}$.

Si un retrait de valeur dans une variable (`ntree` ou une variable décrivant un nœud du graphe) impliqué dans la contrainte `tree` mène a un domaine vide il est nécessaire de revenir dans un état antérieur consistant. Pour ce faire, il est possible d'utiliser les structures de données *backtrackables* fournies par le solveur de contraintes utiliséÈ [5]. L'implémentation initiale de la contrainte `tree` utilisait ces structures de données afin d'enregistrer dynamiquement et restaurer les propriétés de graphe comme les composantes fortement connexes et les nœuds dominants du graphe orienté $\mathcal{G}$. Ainsi, l'utilisation de ces structures de données backtrackables dédiées fait que la contrainte `tree` initiale est dépendante du solveur.

L'état de l'art classique des algorithmes de graphe propose plusieurs algorithmes incrémentaux permettant de maintenir les propriétés de graphe impliquées dans la contrainte `tree` comme les composantes fortement connexes et la fermeture transitive [9], ou les nœuds dominants [19]. Deux questions demeurent : est-il vraiment nécessaire d'utiliser des structures de données backtrackables lorsque des algorithmes incrementaux en ajout et en retrait existent ? Si aucune structure backtrackable n'est utilisée (c.-à.-d. qu'il n'est pas nécessaire de *trailer* les modifications des structures de données utilisées dans la contrainte),

[2] Une *racine potentielle* d'un graphe orienté $\mathcal{G}$ est un nœud qui peut potentiellement être racine d'un arbre dans une solution satisfaisant la contrainte `tree`.

[3] Le *graphe réduit* $\mathcal{G}_r$ est dérivé d'une graphe orienté donné $\mathcal{G}$ en associant à chaque composante fortement connexe, un nœud de $\mathcal{G}_r$ et à chaque arc de $\mathcal{G}$ qui connecte différentes composantes fortement connexes, correspond un arc dans $\mathcal{G}_r$.

[4] Étant donné un graphe orienté $\mathcal{G}$, un nœud j *domine* un nœud k par rapport à un nœud i si et seulement si n'importe quel chemin de i à k atteint le nœud j avant le nœud k

[5] Le solveur Choco propose plusieurs structures *backtrackables* utilisant des entiers, des booléens et des bitsets. Ils sont basés sur une approche de *trailing* [18] qui enregistre, pour chaque événement modifiant la structure de données, les informations nécessaires pour défaire ses effets.

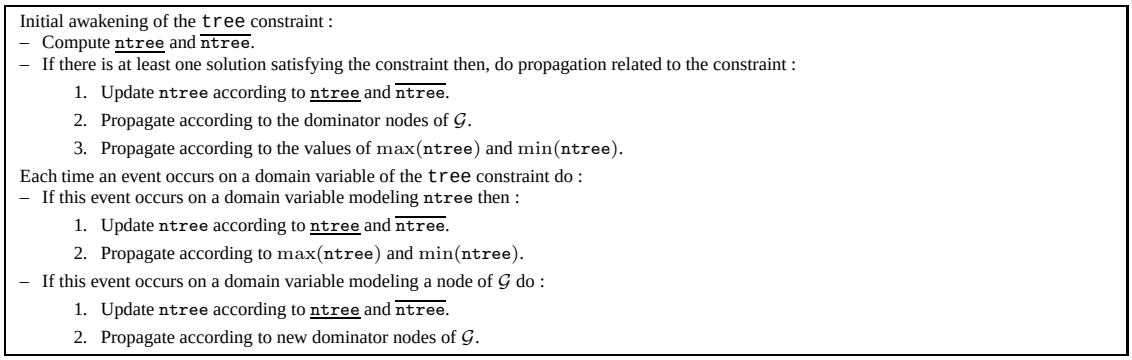


FIG. 3 – Squellete d'implémentation de la contrainte `tree`au sein de *choco*.

quelles sont les relations entre les contraintes et le solveur ?

Le goulot d'étranglement de la complexité de la contrainte `tree` est lié au fait de maintenir de manière répétée plusieurs propriétés de graphe (composantes fortement connexes – cfc, fermeture transitive, nœud dominant) du graphe orienté $\mathcal{G}$ entre deux étapes de recherche. De manière basique, une nouvelle approche implémentant une telle contrainte et respectant la séparation des préoccupations introduite dans la figure 2, peut être décomposée de la manière suivante : premièrement, *l'interface de la contrainte* est décomposée en une structure de la contrainte et en un algorithme de filtrage. La *structure de la contrainte* est basée sur une structure de données générique, incrémentale en ajout et en retrait, qui modélise le graphe orienté $\mathcal{G}$ et ses propriétés associées (c.-à.-d., cfc et la fermeture transitive). Cette partie contient les primitives permettant de mettre à jour la structure de données en fonction des retraits et des ajouts d'arcs. Ces primitives, pour résumer, calculent chaque propriété en ne considérant qu'un graphe partiel nécessaire[6] du graphe d'origine $\mathcal{G}$. *L'algorithme de filtrage*, basé sur les propriétés maintenues par le graphe (représenté par la structure de la contrainte), retire les arcs de $\mathcal{G}$ qui sont inconsistants avec la contrainte `tree`. Deuxièmement, *l'interface du solveur* et le *gestionnaire d'événements* qui assurent une relation bidirectionnelle entre les événements survenant sur le domaine des variables (c.-à-d. retraits/restaurations des valeurs dans les domaines) et des événements survenant dans le graphe orienté $\mathcal{G}$ (retrait/ajout d'arcs).

Dans l'implémentation actuelle de la contrainte `tree` prête à brancher, chaque fois qu'un événement survient dans le domaine d'une variable, cet événement est d'abord interprété par l'interface du solveur pour produire un événement interne. Ensuite, la structure de données interne est mise à jour avec cet événement interne. Alors, l'algorithme de filtrage dédié à la contrainte `tree` est appliqué et les événements internes résultants sont traduits en événements génériques et sont envoyés au gestionnaire d'événements. Nous pouvons maintenant parler du gestionnaire d'événements. À partir des domaines des variables impliquées dans un CSP, généralement, quatre types d'événements peuvent être distingués : le retrait de valeurs dans les domaines, l'instanciation de variables, la mise à jour d'une borne inférieure et la mise à jour d'une borne supérieure. Cependant, les instanciations et les mises à jour de bornes peuvent facilement être décomposées en un ensemble de retraits. Ainsi, pour chaque type d'événement envoyé par l'algorithme de recherche à l'interface du solveur, une traduction de l'événement en un ensemble de retraits dans la structure de la contrainte est réalisée. Cependant, tous les retraits d'arcs ne sont pas gérés de la même manière par le gestionnaire d'événements. En effet, l'événement à l'origine de l'ensemble des retraits est considéré afin d'améliorer l'efficacité des mises à jour de la structure de la contrainte. Par exemple, un ensemble de retraits liés a un événement d'instanciation survenant sur une variable entraîne une modification locale dans le voisinage du nœud correspondant dans le graphe orienté $\mathcal{G}$ associé avec la contrainte. Cette information peut être prise en compte dans le but de réaliser ces modifications efficacement.

5 Évaluation

Dans cette partie, nous nous proposons d'évaluer le comportement des contraintes *prêtes à brancher*. Toutes les expérimentations ont été réalisées avec `choco` (version 1.2.04) et `gecode` (version 1.3.1) sur un processeur Intel Xeon cadencé à 2.4GHz et possédant 1Go de RAM, mais dont seulement 128Mo alloués à la machine virtuelle java.

Cette évaluation est réalisée en deux étapes. La première (section 5.1), évalue le sur-coût engendré par le découplage. Pour cela, nous nous focalisons sur la contrainte `boundAllDiff` dans le contexte des n reines (tableau 1). Puis, la contrainte `tree` montre la portabilité effective de sa version *prête à brancher* pour les solveurs `Choco` et `Gecode` (tableau 2). La deuxième étape (section 5.2), évalue le comportement des contraintes *prêtes à brancher* dans un environnement hétérogène (on retrouve à la fois des

[6] Étant donné un graphe orienté $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, un *graphe partiel* $\mathcal{G}'$ de $\mathcal{G}$ est défini par $(\mathcal{V}' \subseteq \mathcal{V}, \{(i,j) \in \mathcal{E} \mid i,j \in \mathcal{V}'\})$

boundAllDiff		nReines					
		25	50	75	100	150	200
prête à brancher		22	35	1538	2461	5741	14217
ad-hoc		19	35	1531	2435	5711	14006

TAB. 1 – Comparaison des temps de calcul (ms) pour les versions prête à brancher et *ad hoc* de la contrainte `boundAllDiff`.

contraintes *ad hoc* et des contraintes *prêtes à brancher* dans le modèle). Pour cela, nous nous intéressons au problème de chemin hamiltonien (tableau 3).

5.1 Sur-coût lié à la portabilité

Nous évaluons le sur-coût d'interfaçage des contraintes `boundAllDiff` et `tree` avec deux solveurs de contraintes : `Choco` et `Gecode`. L'interface traduit une information liée au solveur en des informations génériques. L'implémentation de cette interface dépend du niveau d'information fourni par le solveur. `Choco`, centré événement, est capable de fournir aux contraintes les modifications ayant provoqué leur réveil. `Gecode`, quant à lui centré propagateur, ne peut fournir cette information. Pour la contrainte `tree` branchée sur `Choco`, l'interface présente une complexité temporelle directement liée au nombre de modifications de domaines réalisées tandis que pour `Gecode`, l'interface doit passer en revue le domaine de chaque variable pour identifier les modifications. Dans le cas de `boundAllDiff`, l'interface n'a uniquement besoin de connaître les valeurs maximale et minimale des domaines des variables aussi bien pour l'un que l'autre des solveurs.

Nous résolvons les n reines à 25, 50, 75, 100, 150 and 200 reines. Le problème est modélisé à l'aide de 3 contraintes `boundAllDiff` (en version prête à brancher d'une part et en version *ad hoc* d'autre part). Cette contrainte ne gère aucune structure de données interne, ainsi le sur-coût pour la version prête à brancher est minimal. Pour 200 reines, le temps passé dans les trois modules d'interface (une interface par contrainte) ne représente que 1.5% du temps global de calcul (tableau 1).

On peut noter qu'une contrainte *prête à brancher* peut-être utilisée avec n'importe quel problème. Le tableau 2 montre les détails du temps de calcul utilisé par les différentes composantes de la contrainte `tree`, tant pour `Choco` que pour `Gecode`. Pour des graphes d'ordre$\{25, 50, 75, 100, 150, 200\}$, et de densité $\{0.05, 0.2, 0.4, 0.5, 0.6, 0.8, 0.95\}$, 50 instances d'un problème de partionnement de graphe sont générées (soit au total 2100 graphes). Notons que les deux solveurs utilisent ici la même heuristique de choix de variable. La colonne "Interface" du tableau 2 montre parfaitement le sur-coût engendré par l'interface dans Gecode. On note aussi ce que le temps passé dans la contrainte elle-même est le même dans les deux solveurs : c'est ce que montrent les colonnes "Contrainte" et "Structure".

5.2 Comportement en pratique

Cette section résume les résultats expérimentaux des contraintes prêtes à brancher utilisées pour résoudre un problème de chemin hamiltonien. Cette évaluation montre l'efficacité des contraintes prêtes à brancher dans un contexte pratique. Pour cela, le problème de chemin hamiltonien est modélisé à l'aide des contraintes `boundAllDiff` et `tree`.

L'évaluation est réalisée en observant toutes les combinaisons possibles (ad hoc/prête à brancher) des contraintes `tree` et `boundAllDiff` avec le solveur choco. Pour chaque ordre de graphe dans $\{25, 50, 75, 100, 150, 200\}$, et pour des densités $\{0.10, 0.25, 0.40, 0.50, 0.65, 0.75, 0.90\}$, 50 instances sont générées (soit globalement 2100 graphes). Ici, un timeout de 5 minutes a été fixé et une heuristique aléatoire de choix de variable a été utilisée.

Les résultats présentés dans le tableau 3 montrent qu'utiliser des contraintes prêtes à brancher n'a pas d'impact sur l'efficacité, même en cas de combinaison avec des contraintes *ad hoc*. Même, on peut voir que la version prête à brancher de la contrainte `tree` est bien meilleure que la version *ad hoc*. Ceci est principalement dû à la nature complètement incrémentale de cette contrainte (aucune structure backtrackable n'est utilisée). Enfin, la contrainte `boundAllDiff` dans sa version prête à brancher est équivalente à sa version *ad hoc*.

6 Discussion

Cet article tente d'identifier les limites du découplage d'une contrainte globale du solveur sous-jacent. Nous avons montré dans les paragraphes précédents qu'il n'y a pas de frein majeur à un tel découplage. En fait, cette question revient à lever deux interrogations :

6.1 Est-ce que l'implémentation d'une contrainte prête à brancher est difficile ?

Les contraintes globales sont généralement implémentées à l'aide des structures dédiées et de l'API d'un solveur donné. Ces contraintes, le plus souvent, embarquent des structures de données persistantes utilisées pour maintenir leur propre niveau de consistance pendant la recherche de solution. En réalité, cette interférence du solveur avec l'architecture interne de la contrainte est plutôt une limitation pour l'efficacité et l'expressivité d'une implémentation. Par exemple, les nouveautés proposées par les langages ne sont pas, la plupart du temps, directement utilisables étant donné le temps de latence induit par le solveur lui-même (par exemple, les *generics* en Java).

Ordre du graphe	temps Choco (ms)			temps Gecode (ms)		
	Interface	Contrainte	Graphe	Interface	Contrainte	Structure
25	5	10	27	6	10	27
50	5	57	229	24	56	228
75	11	190	863	58	186	879
100	18	440	2287	120	439	2310
150	50	1466	9524	368	1329	9596
200	82	3214	27551	812	3009	27097

TAB. 2 – Temps de calcul pour `tree` branchée sur les deux solveurs (Choco et Gecode).

Ordre du graphe	contrainte `tree` ad hoc		contrainte `tree` à brancher	
	ad hoc `boundAllDiff`	à brancher `boundAllDiff`	ad hoc `boundAllDiff`	à brancher `boundAllDiff`
25	57	58	56	52
50	5529	5577	6224	6235
75	10858	10745	10311	10126
100	19201	19267	14956	14855
150	60683	60612	27271	27380
200	156055	155793	45099	45919

TAB. 3 – Résolution du problème de chemin hamiltonien combinant contraintes ad hoc et prêtes à brancher (ms).

Ainsi, dans le contexte des contraintes prêtes à brancher la question des structures de données internes n'en est plus une car toute latitude est laissée au développeur. Par contre, la vraie question est la manière de mettre à jour la représentation interne de la contrainte face aux événements fournis par le solveur. Une réponse satisfaisante doit prendre en compte la complexité du maintien des propriétés maintenues par la représentation interne. Intuitivement, trois cas peuvent être distingués : dans le premier cas, les propriétés sont calculées à chaque appel de la contrainte sans aucun effet mémoire (comme dans la section 4.2). Ainsi, après chaque modification les propriétés sont recalculées et la contrainte peut les exploiter ; dans le deuxième cas, les propriétés ne peuvent être recalculées de manière efficace mais il existe des algorithmes complètement incrémentaux qui permettent de prendre en compte les modifications (cas de la section 4.3). Dans le cas des propriétés de graphe, par exemple, de nombreux travaux existent pour prendre en compte dynamiquement des modifications de structures. Ainsi, [17] propose un algorithme de maintien d'un arbre recouvrant de poids minimal devant des ajouts/retraits d'arêtes ; dans le dernier cas, il est nécessaire d'embarquer dans la contrainte des mécanismes explicites de retour arrière (selon diverses méthodes : trailing, copie, recalcul). Là, il est tout de même nécessaire d'être capable de savoir depuis la contrainte si un retour arrière a eu lieu ou non.

6.2 Est-ce que réaliser l'interface pour une contrainte prête à brancher est difficile ?

Les solveurs manipulent des événements sur les variables décrivant des modifications des domaines : retraits de valeur(s), mise à jour de borne(s), etc. Ces événements sont compris et traités par toutes les contraintes portant sur les variables du solveur. Mais, pour les contraintes prêtes à brancher, ces événements sont généralement vides de sens *per se*. C'est pourquoi l'interface doit les traiter pour transformer ces événements du solveur en événements sur les structures de données internes de la contrainte.

Dans la première approche de découplage, les événements du solveur peuvent traduit en *quelque chose a changé*. Cette information permet de recalculer les propriétés à maintenir.

Dans la seconde approche, les événements du solveur sont transformés en des mises à jour de la structure de données interne traitées ensuite par les algorithmes incrémentaux fournis par la contrainte. Ces mises à jour dépendent du type de modifications que sont capables de traiter les algorithmes en question. Par exemple, dans la section 4.3, les événements du solveur ont été transformés en des retraits (ou ajouts) d'arêtes dans le graphe de référence.

Enfin dans la dernière approche, les événements du solveur sont transformés en des événements compris par la structure backtrackable générique utilisée.

La limite principale au découplage des contraintes globale n'est pas une question de génie logiciel mais plutôt une question d'interopérabilité des langages de programmation utilisés pour implémenter les contraintes. Par exemple, les contraintes de choco sont développées en Java, celles de Gecode en C++, celles de Comet en Comet et les contraintes de Sicstus Prolog en C. Évidemment des outils existent pour traiter ces problématiques. Par exemple, nous avons utilisé JNI (Java Native Interface) pour interfacer nos contraintes prêtes à brancher avec Gecode (à travers le module Gecode/J). JNI permet à du code Java tournant sur une machine virtuelle d'appeler et d'être appelé par d'autres applications écrites dans d'autres langages de programmation. Les coûts engendrés par une telle interface supplémentaire sont liés au type d'arguments pas-

sés entre les différent langages. Mais, en faisant attention, les applications Java peuvent appeler du code natif de manière relativement efficace. De plus, le sur-coût de JNI devient négligeable quand le code natif est fort consommateur de cpu.

Dans le cas de notre interface, les paramètres sont des messages d'événements (sur les variables) ainsi JNI est une solution envisageable. Par contre, pour une application codée en C, appeler une fonction Java est peu efficace car il est alors nécessaire d'utiliser des mécanismes réflexifs dans Java [23][7]. Il existe d'autres approches pour assurer l'interopérabilité des langages : communication par TCP/IP ou par IPC (inter-process communication) par exemple. Les applications Java en particulier peuvent utiliser les technologies objet distribuées comme l'API IDL[8].

7 Conclusion

Cet article s'est proposé d'étudier les liens entre les contraintes et les solveurs dans lesquels elles sont implémentées pour montrer qu'un découplage est généralement possible. Un tel découplage permet d'utiliser des contraintes globales complexes (assez difficiles à implémenter) avec différents solveurs. Nous avons montré qu'une limitation reposait sur la gestion de la mémoire pour le retour arrière tant sur les domaines des variables que pour les structures de données internes manipulées par les contraintes. Nous avons proposé une discussion sur la manipulation d'une mémoire distribuée pour les structures de données de chaque contrainte. En particulier, nous avons montré que dans le cas de la contrainte `tree` et plus généralement quand des algorithmes complètement incrémentaux existent, un tel découplage est efficace.

De plus, pour une contrainte globale donnée, un découplage entre filtrage et structure de données interne est un point central qui permet une véritable séparation des préoccupations (importante d'un point de vue génie logiciel). En effet, améliorer un algorithme de filtrage est différent d'améliorer la gestion des structures de données. Mais, on se rend compte que pour beaucoup de contraintes c'est ce qui fait la différence (en particulier pour les contraintes sur les graphes comme la contrainte `tree`). Enfin, nous montrons que le frein principal au découplage des contraintes globales n'est pas uniquement un problème de génie logiciel mais reste au niveau de l'interopérabilité des langages de programmation.

Nos travaux futurs concernent la mise à disposition d'un ensemble de contraintes globales sur les graphes et sur un ensemble d'interfaces pour gérer efficacement l'intéropérabilité entre langage (avec une focalisation sur Java et C/C++). Bien sûr, toutes les contraintes d'un système n'ont pas nécessairement vocation à être découplées (par exemple, les contraintes arithmétiques ou toute contrainte ne nécessitant pas le maintien d'un état interne). L'objectif à long terme de cette réflexion est la promotion de la notion de contrainte globale en dehors du strict cadre de la programmation par contraintes classique en en proposant l'utilisation, par exemple, dans des stratégies de recherche locale ou des solveurs à base d'explications.

[7] `http://java.sun.com/docs/books/performance/1st_edition/html/JPNativeCode.fm.html`

[8] `http://java.sun.com/docs/books/jni/html/intro.html#1811`

Références

[1] N. Beldiceanu, P. Flener, and X. Lorca. The *tree* constraint. In *CP-AI-OR'05*, volume 3524 of *LNCS*, pages 64–78, 2005.

[2] N. Beldiceanu, X. Lorca, and P. Flener. Combining tree partitioning, precedence, and incomparability constraints. *Constraints*, 13(4), 2008.

[3] Nicolas Beldiceanu, Mats Carlsson, Sophie Demassey, and Thierry Petit. Global constraint catalogue : Past, present and future. *Constraints*, 12(1) :21–62, 2007.

[4] Nicolas Beldiceanu, Mats Carlsson, and Thierry Petit. Deriving filtering algorithms from constraint checkers. In *CP'04*, volume 3258 of *LNCS*, pages 107–122, 2004.

[5] Nicolas Beldiceanu, Mats Carlsson, Emmanuel Poder, R. Sadek, and Charlotte Truchet. A generic geometrical constraint kernel in space and time for handling polymorphic *k*-dimensional objects. In *CP'07*, LNCS, pages 180–194, 2007.

[6] C. Bessière and P. van Hentenryck. To Be or Not to Be... a Global Constraint. In *CP'03*, LNCS, pages 789–794, 2003.

[7] Christian Bessière, Emmanuel Hebrard, Brahim Hnich, Zeynep Kiziltan, and Toby Walsh. The range and roots constraints : Specifying counting and occurrence problems. In *IJCAI*, pages 60–65, 2005.

[8] Pham Quang Dung, Y. Deville, and P. van Hentenryck. Ls(graph) : A local search framework for constraint optimization on graphs and trees. In *SAC*, March 2009.

[9] D. Eppstein, Z. Galil, and G. Italiano. *Dynamic graph algorithms*. CRC Press, 1997.

[10] Narendra Jussien and Vincent Barichard. The PaLM system : explanation-based constraint programming. In *Proceedings of TRICS workshop of CP*, pages 118–133, Singapore, September 2000.

[11] Mikael Z. Lagerkvist and Christian Schulte. Advisors for incremental propagation. In *CP'07*, LNCS, pages 409–422, 2007.

[12] J-L. Laurière. A Language and a Program for Stating and Solving Combinatorial Problems. *Artificial Intelligence*, 10 :29–127, 1978.

[13] Alejandro López-Ortiz, Claude-Guy Quimper, John Tromp, and Peter van Beek. A Fast and Simple Algorithm for Bounds Consistency of the AllDifferent Constraint. In *IJCAI*, pages 245–250, 2003.

[14] R. Mohr and T.C. Henderson. Arc and path consistency revisited. *Artif. Intell.*, 28(2) :225–233, 1986.

[15] J.-C. Régin. A filtering algorithm for constraints of difference in CSP. In *AAAI'94*, pages 362–367, 1994.

[16] Jean-Charles Régin. Maintaining arc consistency algorithms during the search without additional space cost. In *CP*, volume 3709 of *LNCS*, pages 520–533, 2005.

[17] Jean-Charles Régin. Simpler and incremental consistency checking and arc consistency filtering algorithms for the weighted spanning tree constraint. In *CPAIOR'08*, LNCS, pages 233–247, 2008.

[18] Christian Schulte. Comparing Trailing and Copying for Constraint Programming. In *ICLP'99*, pages 275–289, 1999.

[19] V. C. Sreedhar, G. R. Gao, and Y.-F. Lee. Incremental Computation of Dominator Trees. *ACM Transactions on Programming Languages and Systems*, 19(2) :239–252, March 1997.

[20] P. van Hentenryck. *Constraint satisfaction in logic programming*. MIT Press, 1989.

[21] Pascal van Hentenryck and Laurent Michel. Growing COMET. In Frédéric Benhamou, Narendra Jussien, and Barry O'Sullivan, editors, *Trends in Constraint Programming*, chapter 17, pages 291–297. ISTE, 2007.

[22] W. J. van Hoeve. The alldifferent constraint : A survey. *CoRR*, cs.PL/0105015, 2001.

[23] Steve Wilson and Jeff Kesselman. *Java Platorm Performance : Strategies and Tactics*. Addison-Wesley, 2000.

Séquencer et compter avec la contrainte multicost-regular

Julien Menana **Sophie Demassey**

École des Mines de Nantes, LINA CNRS UMR 6241, F-44307 Nantes, France.

{julien.menana,sophie.demassey}@emn.fr

Abstract

Ce papier introduit une contrainte globale associant une contrainte `regular` et plusieurs coûts cumulatifs. Le problème d'optimisation sous-jacent à la contrainte `multicost-regular` est NP-difficile mais peut être réduit par une relaxation lagrangienne performante. L'expressivité et l'efficacité de cette nouvelle contrainte sont évaluées sur des problèmes de planification de personnel basés sur des réglementations du travail réelles. Les résultats empiriques de la contrainte `multicost-regular` surpassent de façon significative les résultats donnés par un modèle décomposé comprenant d'une part la contrainte `regular` et d'autre part la contrainte `global-cardinality`.

1 Introduction

L'action simultanée du séquencement et du comptage d'objets intervient dans de nombreux problèmes de décision combinatoires, et notamment dans les problèmes de planification d'horaires et de tournées de véhicules. Lors de sa tournée, un véhicule visite une suite de lieux en empruntant un chemin dans le réseau routier. Ce chemin ou circuit est défini selon des critères numériques tels que la distance totale parcourue, le temps nécessaire, ou la capacité du véhicule. Si un seul attribut numérique est spécifié, trouver un circuit possible revient à résoudre un problème de plus court/long chemin. Lorsque plusieurs attributs sont renseignés, le problème de plus court/long chemin sous contraintes de ressources – ou Resource Constrained Shortest/Longest Path Problem (RCSPP) –devient NP-complet. Toutes ces conditions numériques restreignent le nombre de chemins pouvant correspondre au sein du réseau routier. C'est pourquoi il est plus efficace de propager l'ensemble de ces critères pendant la recherche de chemins, plutôt que de les prendre en compte séparément. Un parallèle peut être fait avec les problèmes de planification d'horaires. En effet, planifier le travail d'un employé revient à définir, sur une plage de temps donnée, une séquence d'activités (ou postes) respectant différentes règles, nombreuses et variées, comme par exemple : « une nuit de travail est suivie d'une matinée de repos », un travail de nuit est payé double, au moins dix jours de repos par mois, etc. Ainsi, la définition d'un planning horaire mêle étroitement des critères structurels – les séquences d'activités autorisées ou interdites – et des critères numériques – les coûts et les compteurs d'activités. La modélisation de toutes ces conditions individuellement est en soi une tâche difficile, pour laquelle l'expressivité et la flexibilité de la programmation par contraintes sont reconnues. Les modéliser de manière efficace est plus difficile encore puisque cela implique de pouvoir raisonner sur l'ensemble des conditions de manière globale. En introduisant la contrainte globale `regular`, Pesant [9] proposait une méthode élégante et efficace pour modéliser et pour résoudre les contraintes structurelles de séquencement comme une recherche de chemins dans un graphe acyclique. Les contraintes d'optimisation `soft-regular` [14] et `cost-regular` [4] étendent cette approche en considérant des bornes sur le coût total (coût de violation ou coût financier) d'une séquence d'activités. Le problème sous-jacent consiste alors à calculer les plus courts et plus longs chemins du graphe acyclique et à les filtrer selon ces bornes. La contrainte `cost-regular` a été appliquée avec succès, au sein d'une approche de génération de colonnes basée sur la programmation par contraintes, pour résoudre des problèmes réels de planification de personnel [4]. Pourtant, le modèle de programmation par contraintes utilisé révèle une trop

faible interaction entre la contrainte `cost-regular` et une contrainte externe `global-cardinality` chargée de limiter le nombre d'occurrences des activités. En effet, avec une telle décomposition, le graphe support de `cost-regular` conserve de nombreux chemins qui ne satisfont pas aux contraintes de cardinalité. Dans cet article, nous poursuivons la généralisation de cette approche en proposant de traiter plusieurs compteurs ou attributs de coûts au sein d'une seule contrainte globale `multicost-regular`. Celle-ci permet de raisonner simultanément sur les critères de séquencement, de coûts et d'occurrences. Comme évoqué ci-avant, le problème d'optimisation sous-jacent est un RCSPP et il reste NP-difficile même quand le graphe est acyclique. Ainsi, l'algorithme de filtrage que nous proposons atteint un niveau de consistance relâché. Celui-ci s'appuie sur une relaxation lagrangienne du RCSPP en suivant le principe de filtrage par relaxation lagrangienne défini par Sellmann [11]. Notre implémentation de `multicost-regular` est disponible dans la distribution du solveur de contraintes open-source *CHOCO* [1].

Ce papier est organisé de la façon suivante. Dans la Section 2, la classe des contraintes `regular` est présentée et une comparaison théorique entre l'approche de recherche par chemins de Pesant [9] et l'approche par décomposition proposée par Beldiceanu et al. [1] est développée. Nous introduirons alors la nouvelle contrainte `multicost-regular`. Dans la Section 3, une introduction à l'algorithme de filtrage par relaxation lagrangienne est présentée. Dans la Section 4, une variété de règles de travail standards sont décrites et l'étude d'une méthode systématique de création d'une instance `multicost-regular` à partir d'un ensemble de ces règles est présentée. Enfin, dans la Section 5, les résultats de calculs empiriques sur des cas tests de problèmes de planification de personnel sont comparés. Ils montrent l'atout significatif de `multicost-regular` comparée à un modèle par décomposition avec des contraintes `regular` et `globalcardinality`.

2 Contraintes d'appartenance à un langage rationnel

Cette section revient sur la définition de la contrainte `regular` et les travaux associés puis introduit la contrainte `multicost-regular`. Tout d'abord, nous rappelons les notions de base de la théorie des automates et introduisons les notations qui seront utilisées dans ce papier :

Nous considérons un ensemble non-vide Σ appelé *alphabet*. Les éléments de l'ensemble Σ sont appelés *lettres*, les séquences de lettres sont appelées *mots*, et les ensembles de mots sont appelés *langages* dans Σ. Un *automate* Π est un multigraphe orienté (Q, Δ) dont les arcs sont étiquetés par les lettres d'un alphabet Σ, et où deux sous-ensembles non-vides de noeuds I et A sont distingués. L'ensemble de noeuds Q est appelé l'ensemble d'*états* de Π, I est l'ensemble des *états initiaux*, et A est l'ensemble des*états finaux*. L'ensemble non-vide $\Delta \subseteq Q \times \Sigma \times Q$ des arcs est appelé l'ensemble des *transitions* de Π. Un mot dans Σ est dit *accepté* par Π s'il correspond à la séquence d'étiquettes des arcs d'un chemin depuis un état initial vers un état final dans Π. L'automate Π est un *automate fini déterministe (AFD)* si Δ est fini et s'il possède un unique état initial ($I = \{s\}$) et aucune paire de transitions partageant la même extrémité initiale et la même étiquette. Le langage accepté par un AFD est un *langage rationnel* ou *régulier*.

2.1 Chemins et décomposition : deux approches pour `regular`

La contrainte `regular` a été introduite par Pesant [9]. Étant donnée une séquence $X = (x_1, x_2, ..., x_n)$ de variables et un automate fini déterministe $\Pi = (Q, \Sigma, \Delta, \{s\}, A)$, la contrainte `regular`$(X, \Pi)$ est vérifiée si et seulement si X est un mot de taille n sur Σ accepté par Π. Par définition, les solutions de `regular`(X, Π) correspondent une à une aux chemins de n arcs connectant s à un sommet dans A dans le multigraphe orienté Π. Soit $\delta_i \in \Delta$ l'ensemble des transitions qui correspondent au i-ème arc d'un tel chemin, alors une valeur pour x_i est cohérente si et seulement si δ_i contient une transition étiquetée par cette valeur.

Incidemment, Pesant [9] et Beldiceanu et al [1] ont introduit deux approches orthogonales pour assurer la Consistance d'Arc Généralisée (CAG) de la contrainte `regular` (voir la figure 1). L'approche proposée par Pesant [9] est de développer Π comme un AFD acyclique Π_n qui accepte uniquement les mots de longueur n. Par construction, Π_n est un multigraphe à plusieurs couches avec l'état s dans la couche 0 (la source), les états finaux A dans la couche n (les puits), et où l'ensemble des arcs d'une couche quelconque i coïncide avec δ_i. Une première recherche étendue permet de maintenir la cohérence entre Π_n et les domaines des variables en retirant les arcs dans δ_i dont les étiquettes ne sont pas dans le domaine de x_i, puis en retirant les noeuds et arcs qui ne sont pas connectés à une source et à un puits. Dans Beldiceanu et al [1], une contrainte `regular` est décomposée en n contraintes ternaires définies en extension et modélisant les ensembles $\delta_1, \delta_2, \ldots, \delta_n$. La décomposition introduit des variables d'état $q_0 \in \{s\}$, $q_1, \ldots, q_{n-1} \in Q$, $q_n \in A$ et des contraintes ternaires de transition

[1] `http ://choco.emn.fr/`

$(q_{i-1}, x_i, q_i) \in \delta_i$. Le réseau des contraintes de transition étant Berge-acyclique, appliquer une consistance d'arc sur la décomposition permet d'atteindre la CAG sur `regular`.

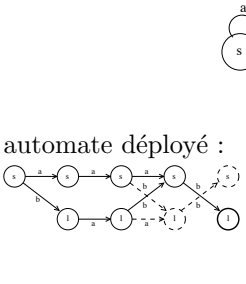


FIG. 1 – Soit l'AFD décrit ci-dessus appliqué à $X \in \{a, b\} \times \{a\} \times \{a, b\} \times \{b\}$. L'automate déployé de `regular` est présenté à gauche et le modèle décomposé sur la droite du schéma. Les transitions en pointillé sont filtrées dans les deux modèles.

Dans la première approche, un algorithme spécifique est défini pour maintenir l'ensemble des chemins support, tandis que la seconde approche permet de laisser le solveur propager les contraintes de transitions. Ces deux approches de filtrage sont orthogonales mais, selon la propagation du second modèle, elles peuvent procéder de manière rigoureusement identique.

Assumons, sans perte de généralité, que Σ est l'union de domaines variables, alors l'exécution initiale de l'algorithme de Pesant pour la construction de Π_n est réalisé en $O(n|\Delta|)$ en temps et en espace (avec $\Delta \leq |Q||\Sigma|$ si Π est un AFD). Le filtrage incrémental procède par un parcours avant/arrière de Π_n et possède cette même complexité de pire cas. En réalité, la complexité de l'algorithme dépend plus de la taille $\Delta_n|$ de l'automate déployé Π_n que de la taille $|\Delta|$ de l'automate spécifié Π. Notons par exemple que lorsque l'automate spécifié Π accepte uniquement des mots de taille n alors il est déjà déployé ($\Pi = \Pi_n$) et la première exécution de l'algorithme est en $O(|\Delta|)$. En pratique, de même que dans nos tests (Section 5), Π_n peut même être nettement plus petit que Π. Cela signifie que de nombreux états finaux dans Π ne peuvent pas être atteints en exactement n transitions. Le filtrage s'effectue alors en $O(|\Delta_n|)$ avec ici $|\Delta_n| \ll n|\Delta|$.

`regular` est une contrainte très expressive. Elle est utile pour modéliser les séquencements acceptés (motifs), contraintes fréquentes dans les problèmes de planification. Elle permet aussi de reformuler d'autres contraintes globales [1] ou encore de modéliser des contraintes définies en extension. Une autre application de `regular` est de modéliser une contrainte glissante : Récemment, Bessière et al. [2] ont introduit la méta-contrainte `slide`. Dans sa forme la plus générale, `slide` prend comme arguments une matrice de variables Y de taille $n \times p$ et une contrainte C d'arité pk avec $k \leq n$. `slide`(Y, C) est satisfaite si et seulement si $C(y^1_{i+1}, \ldots, y^p_{i+1}, \ldots, y^1_{i+k}, \ldots, y^p_{i+k})$ est satisfaite pour tout $0 \leq i \leq n - k$. En utilisant la décomposition proposée dans [1], `regular`(X, Π) peut être reformulée comme `slide`$([Q, X], C_\Delta)$, avec Q la séquence de variables d'état et C_Δ la contrainte de transition $C_\Delta(q, x, q', x') \equiv (q, x, q') \in \Delta$. Inversement [2], une contrainte `slide` peut être reformulée comme une contrainte `regular` mais cela nécessite d'énumérer tous les n-uplets valides de C. Cette reformulation peut cependant s'avérer utile dans le cadre de la planification (notamment en car-scheduling) pour modéliser une contrainte de cardinalité glissante, aussi connue sous le nom de `sequence`. De puissants algorithmes spécialisés existent pour cette contrainte (voir e.g. [7]). L'avantage de la reformulation est qu'elle permet d'intégrer l'automate résultant avec d'autres règles de motifs comme nous le montrerons dans la Section 4. Enfin, notons l'existence de travaux liés aux contraintes dans les grammaires à contexte non-contraint (voir e.g. [6]) bien que les langages rationnels soient généralement suffisants dans le cadre de la planification de personnel.

2.2 Bornes de coûts et cardinalités

Les problèmes de planification de personnel sont en général définis comme des problèmes d'optimisation. La plupart du temps, le critère à optimiser est un *coût cumulatif*, c'est à dire la somme des coûts associés à chacune des activités d'un travailleur. Un tel coût peut avoir différentes significations : il peut représenter un coût financier, une préférence, ou un compteur d'actions. Ainsi, concevoir un emploi du temps valide pour un travailleur consiste à définir une séquence d'activités respectant des motifs donnés et dont le coût est borné. Cela peut être spécifié à l'aide d'une contrainte `cost-regular` [4]. Soit $c = (c_{ia})_{i \in [1..n] \times a \in \Sigma}$ une matrice de coûts des activités et $z \in [\underline{z}, \overline{z}]$ une variable bornée ($\underline{z}, \overline{z} \in \mathbb{R}$), `cost-regular`$(X, z, \Pi, c)$ est satisfaite si et seulement si `regular`(X, Π) est satisfaite et $\sum_{i=1}^{n} c_{ix_i} = z$. On notera que la contrainte `knapsack` [13] est un cas spécial. À moins que $P = NP$, la CAG de la contrainte `knapsack` ne peut être assurée au mieux qu'en temps pseudo-polynomial (polynomial en les valeurs des bornes de z). Par conséquent, assurer la CAG de `cost-regular` est NP-difficile.

La définition de `cost-regular` révèle une décomposition naturelle en une contrainte `regular` reliée à une contrainte `knapsack`. En réalité, cette décomposition est équivalente à celle proposée par Beldiceanu

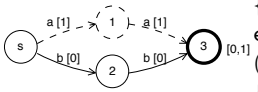
et al. [1] lorsque l'on s'intéresse à un coût cumulatif[2] : des variables de coûts k_i sont associées aux variables d'état q_i, avec $k_0 = 0$ et $k_n = z$, et plusieurs fonctions arithmétiques et contraintes `element` modélisent les `knapsack` et les contraintes de transition. Cette formulation peut être ré-écrite comme $\texttt{slide}([Q, X, K], C^c_\Delta)$, avec $C^c_\Delta(q_{i-1}, x_{i-1}, k_{i-1}, q_i, x_i, k_i) \equiv (q_{i-1}, x_{i-1}, q_i) \in \Delta \wedge k_i = k_{i-1} + c_{ix_i}$. En fonction de la taille des domaines des variables de coûts, la CAG peut être assurée sur `knapsack` dans un temps raisonnable. Cependant, comme l'hypergraphe des contraintes du modèle décomposé n'est plus Berge-acyclique mais α-acyclic, on doit assurer la consistance deux-à-deux des variables partagées – une paire (q_i, k_i) d'état et variable de coût – par les contraintes de transition afin d'atteindre la CAG du modèle complet. Une option similaire proposée pour `slide` [2] est d'assurer l'AC sur l'encodage dual de l'hypergraphe des contraintes C^c_Δ, mais là-aussi cela nécessite d'expliciter tous les n-uplets supports, rendant son utilisation non fonctionnelle.

L'algorithme de filtrage présenté dans [4] pour `cost-regular` est une légère adaptation[3] de l'algorithme de Pesant pour `regular`. Il s'appuie sur le calcul de plus court et plus long chemins dans le graphe déployé Π_n évalué par les coûts des transitions. À chaque noeud (i, q) d'une couche i de Π_n sont associés deux variables de coûts bornées k^-_{iq} et k^+_{iq}, modélisant les longueurs des chemins, respectivement depuis la couche 0 jusqu'à la couche (i, q) et depuis (i, q) jusqu'à la couche n. Les variables de coûts peuvent être initialisée de façon triviale pendant la construction de Π_n : k^-_{iq} dans le sens avant et k^+_{iq} dans le sens arrière. Les bornes de la variable z sont alors réduites selon la condition $z \subseteq k^+_{0s}$. Inversement, un arc $((i-1, q), a, (i, q')) \in \delta_i$ peut être retiré dès que :

$$\underline{k^-_{(i-1)q}} + c_{ia} + \underline{k^+_{iq'}} > \overline{z} \quad \text{or} \quad \overline{k^-_{(i-1)q}} + c_{ia} + \overline{k^+_{iq'}} > \underline{z}.$$

Comme le graphe Π_n est acyclique, surveiller les variables de coût, c'est à dire les plus court et plus long chemins, peut être réalisé par l'algorithme de parcours avec la même complexité $O(|\Delta_n|)$ que pour maintenir la connexité du graphe dans `regular`.

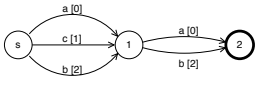
Comme évoqué précédemment, cet algorithme atteint un niveau hybride de consistance sur `cost-regular`. En fait, il assure pour le modèle décomposé une consistance entre les variables d'état et les bornes de la variable de coût associée selon la relation $q_i = (i, q) \iff k_i = k^-_{iq}$. L'algorithme supplante donc le modèle décomposé `knapsack`∧`regular` quand seule la consistance aux bornes est appliquée sur les variables de coûts. Sinon, si l'AC est assurée sur `knapsack` alors les deux approches sont incomparables, comme nous le montrent les deux exemples présentés en Figures 2 et 3.



$x_1 \in \{a, b\}$,
$x_2 \in \{a, b\}$,
$z \in [0, 1]$.

$(q_0, x_1, q_1, j_1) \in \{(s, a, 1, 1), (s, b, 2, 2)\}$,
$\texttt{element}(k_1, j_1, (k_0 + 1, k_0))$,
$(q_1, x_2, q_2, j_2) \in \{(1, a, 3, 1), (2, b, 3, 2)\}$,
$\texttt{element}(z, j_2, (k_1 + 1, k_1))$,
$q_0 \in \{s\}, q_1 \in \{1, 2\}, q_2 \in \{3\}$,
$k_0 \in \{0\}, k_1 \in \{0, 1\}$,
$x_1 \in \{a, b\}, x_2 \in \{a, b\}, z \in \{0, 1\}$.

FIG. 2 – Soit l'AFD représenté avec les coûts entre crochets appliqué à $X = (x_1, x_2) \in \{a, b\} \times \{a, b\}$ et $z \in [0, 1]$. L'algorithme `cost-regular` (sur la gauche) filtre les arcs en pointillé et atteint ainsi la CAG. L'arc-consistance sur le modèle décomposé (sur la droite) n'atteint pas la consistance globale.



$x_1 \in \{a, b, c\}$,
$x_2 \in \{a, b\}$,
$z \in [2, 2]$.

$(x_1, j_1) \in \{(a, 1), (c, 2), (b, 3)\}$,
$\texttt{element}(k_1, j_1, (k_0, k_0 + 2))$,
$(x_2, j_2) \in \{(a, 1), (b, 2)\}$,
$\texttt{element}(z, j_2, (k_1, k_1 + 2))$,
$k_0 \in \{0\}, k_1 \in \{0, 1, 2\}$,
$x_1 \in \{a, b, c\}, x_2 \in \{a, b\}, z \in \{2\}$.

FIG. 3 – Soit maintenant l'AFD représenté ci-dessus appliqué à $X = (x_1, x_2) \in \{a, b, c\} \times \{a, b\}$ et $z \in [2, 2]$. Assurer l'AC sur le modèle décomposé (sur la droite) atteint la CAG. L'algorithme cost-regular (sur la gauche) n'atteint pas la CAG puisque les chemins minimum et maximum traversant l'arc $x_1 = c$ sont consistants avec les bornes de z.

2.3 La contrainte `multicost-regular`

Une généralisation naturelle de `cost-regular` est de traiter plus d'un coût cumulatif : soit un vecteur $Z = (z^0, ..., z^R)$ de variables bornées et $c = (c^r_{ia})^{r \in [0..R]}_{i \in [1..n], a \in \Sigma}$ une matrice de coûts associés aux activités, $\texttt{muticost-regular}(X, Z, \Pi, c)$ est satisfaite si et seulement si $\texttt{regular}(X, \Pi)$ est satisfaite et $\sum_{i=1}^n c^r_{ix_i} = z^r$ pour tout $0 \leq r \leq R$. Une telle généralisation trouve son intérêt dans le cadre de la planification de personnel. En effet, outre le coût financier et les restrictions liées aux motifs, l'emploi du temps individuel est en général assujetti à une contrainte `global-cardinality` limitant le nombre d'occurrences de chaque valeur dans la

[2] Le modèle dans [1] peut traiter non seulement les sommes mais aussi diverses fonctions arithmétiques de coûts, mais aucun exemple de cette utilisation n'est donnée.

[3] Auparavant, l'algorithme était appliqué partiellement – uniquement pour la minimisation – au cas spécial `soft-regular[hamming]` dans [1] et dans [14].

séquence. Ces bornes peuvent réduire drastiquement l'ensemble des solutions valides et donc le graphe support de la contrainte `regular`. Ainsi, traiter les coûts au sein de la contrainte `regular` permet également de réduire la taille du graphe support. Puisqu'il s'agit ici d'une généralisation de la contrainte `cost-regular` ou de la contrainte `global-sequencing` [10], nous ne pouvons pas espérer atteindre la CAG en un temps polynomial. Le modèle de Beldiceanu et al [1] – ou la contrainte `slide` – s'intéresse également à la prise en compte de plusieurs coûts, mais encore une fois le traitement proposé consiste à décomposer une contrainte `regular` en une conjonction de contraintes en extension liées à une contrainte `knapsack` pour chaque coût.

Ainsi, nous proposons d'exploiter la structure du graphe support Π_n pour avoir une bonne propagation relaxée de `muticost-regular`. Les problèmes d'optimisation sous-jacent à `cost-regular` étaient des problèmes de plus court et plus long chemins dans Π_n. Les problèmes d'optimisation sous-jacent à `multicost-regular` sont les Resource Constrained Shortest and Longest Path Problems (RCSPP and RCLPP) dans Π_n. Le RCSPP (resp. RCLPP) a pour but de trouver le plus court (resp. plus long) chemin entre une source et un puits dans un graphe orienté multi-valué, de telle sorte que les quantités de ressources accumulées sur les arcs ne dépassent pas certaines limites. Même pour des graphes acycliques, ce problème est connu pour être NP-difficile[5]. Deux approches sont le plus souvent utilisées pour résoudre RCSPP [5] : la programmation dynamique et la relaxation lagrangienne. Les méthodes basées sur la programmation dynamique étendent les algorithmes usuels de plus court chemin par enregistrement des coûts sur chaque dimension à chaque noeud du graphe. De même que dans `cost-regular`, cela peut aisément être adapté pour le filtrage en convertissant ces étiquettes de coûts en variables de coûts mais la mémoire nécessaire pour l'algorithme serait trop importante. Au lieu de cela nous avons investigué l'approche par relaxation lagrangienne, qui peut aussi être aisément adaptée au filtrage par l'algorithme `cost-regular` sans augmenter de manière drastique les besoins en mémoire.

3 Filtrage basé sur la relaxation lagrangienne

Sellmann [11] a établi le principe de l'utilisation de la relaxation lagrangienne d'un programme linéaire pour filtrer des contraintes d'optimisation. Nous appliquons ce principe au RCSPP/RCLPP pour le filtrage de `multicost-regular`. L'algorithme résultant est un simple procédé itératif dans lequel le filtrage est pris en charge par `cost-regular` dans Π_n pour différentes fonctions de coûts agrégés. Dans cette section, nous présentons le modèle usuel de relaxation lagrangienne pour le RCSPP, ainsi que la méthode de résolution du dual lagrangien basée sur l'algorithme du sous-gradient. Enfin nous montrons comment l'adapter pour le filtrage de `multicost-regular`.

Relaxation lagrangienne du RCSPP. On considère un graphe orienté $G = (V, E)$ muni d'un nœud source s et d'un nœud puits t, et un ensemble de ressources $\mathcal{R}_1, ..., \mathcal{R}_R$. Pour tout $1 \leq r \leq R$, $\overline{z_r}$ (resp. $\underline{z_r}$) dénote la consommation maximum (resp. minimum[4]) de la ressource $\mathcal{R}_\nabla$ sur un chemin de s à t dans G, et c^r_{ij} est la consommation de la ressource $\mathcal{R}_\nabla$ sur tout arc $(i, j) \in E$. Le RCSPP se modélise par le programme linéaire en nombres binaires suivant :

$$\min \sum_{(i,j)\in E} c_{ij}x_{ij} \qquad \text{s.t.} \qquad (1)$$

$$\underline{z^r} \leq \sum_{(i,j)\in E} c^r_{ij}x_{ij} \leq \overline{z^r} \qquad \forall r \in [1..R] \quad (2)$$

$$\sum_{j\in V} x_{ij} - \sum_{j\in V} x_{ji} = \begin{cases} 1 & \text{if } i = s, \\ -1 & \text{if } i = t, \\ 0 & \text{otherwise.} \end{cases} \qquad \forall i \in V \qquad (3)$$

$$x_{ij} \in \{0, 1\} \qquad \forall (i, j) \in E. \quad (4)$$

Dans ce modèle, une variable de décision x_{ij} indique si l'arc (i, j) appartient à un chemin solution. Les contraintes (2) sont les contraintes de ressources et les contraintes (3) sont les contraintes de chemin.

Le principe de la relaxation lagrangienne repose sur le transfert de contraintes dites difficiles dans la fonction objectif où elles sont ajoutées avec un coût de violation $u \geq 0$ appelé *multiplicateur lagrangien*. Le programme linéaire résultant, appelé *sous-problème lagrangien de paramètre u* est une relaxation du problème original. Résoudre le *dual lagrangien* consiste à trouver les multiplicateurs $u \geq 0$ qui fournissent la meilleur relaxation, c.à.d. la plus grande borne inférieure possible.

Les contraintes difficiles du *RCSPP* sont les $2R$ contraintes de ressources (2). En effet, la suppression de ces contraintes produit un problème de plus court chemin qui se résout en temps polynomial. Soit P l'ensemble des solutions $x \in \{0, 1\}^E$ satisfaisant les contraintes (3) : P est l'ensemble des chemins de s à t dans G. Le sous problème lagrangien de paramètre $u = (u_-, u_+) \in \mathbb{R}^{2R}_+$ s'écrit :

$$SP(u) : f(u) = \min_{x\in P} cx + \sum_{r=1}^{R} u^r_+(c^r x - \overline{z^r}) - \sum_{r=1}^{R} u^r_-(c^r x - \underline{z^r})$$

[4] Dans la définition originale du RCSPP, il n'y a pas de borne inférieure sur la capacité : $\underline{z_r}$

Une solution optimale x^u pour $SP(u)$ correspond ainsi à tout plus court chemin dans le graphe $G(u) = (V, E, c(u))$ défini par :

$$c(u) = c + \sum_{r=1}^{R} (u_+^r - u_-^r)c^r \quad (5)$$

et son coût est égal à :

$$f(u) = c(u)x^u + \kappa_u, \quad (6)$$
$$\text{avec } \kappa_u = \sum_{r=1}^{n} (u_-^r \underline{z^r} - u_+^r \overline{z^r}).$$

Résoudre le dual lagrangien. Le dual lagrangien consiste à trouver la meilleur borne inférieure $f(u)$, c.à.d. à maximiser la fonction concave linéaire par morceaux f :

$$LD: \quad f_{LD} = \max_{u \in \mathbb{R}_+^{2R}} f(u) \quad (7)$$

Plusieurs algorithmes permettent de résoudre le dual lagrangien. Nous nous sommes intéressés ici à l'algorithme du sous-gradient [12] pour sa simplicité d'utilisation et parce qu'il ne nécessite pas l'utilisation d'un solveur linéaire. L'algorithme du sous-gradient résout de manière itérative, pour différentes valeurs de u, le sous-problème $SP(u)$. À chaque itération, le vecteur u est mis à jour suivant la direction d'un super-gradient Γ de f avec un pas de longueur μ : $u^{p+1} = max\{u^p + \mu_p \Gamma(u^p), 0\}$. Il existe plusieurs manières de choisir la longueur de pas afin de garantir la convergence vers f_{LD} de l'algorithme du sous-gradient (voir par ex. [3]). Notre implémentation repose sur une longueur de pas standard $\mu_p = \mu_0 \epsilon^p$ avec μ_0 et $\epsilon < 1$ « suffisamment » grands (nous avons choisi empiriquement $\mu_0 = 10$ et $\epsilon = 0.8$). Le super-gradient $\Gamma(u)$ est calculé à partir de la solution optimale $x^u \in P$ retournée par la résolution de $SP(u)$: $\Gamma(u) = ((c^r x^u - \overline{z^r})_{r \in [1..R]}, (\underline{z^r} - c^r x^u)_{r \in [1..R]})$.

De la relaxation lagrangienne au filtrage. Comme montré dans [11], si une valeur est inconsistante dans au moins un des sous-problèmes lagrangiens, alors elle l'est également pour le problème original. C'est la clé du filtrage par relaxation lagrangienne.

Théorème 1. *(i) Soit P un programme linéaire de valeur minimale $f^* \leq +\infty$, soit $\overline{z} \leq +\infty$ une borne supérieure de f^*, et soit $SP(u)$ un sous-problème lagrangien de P de valeur minimale $f(u) \leq +\infty$. Si $f(u) > \overline{z}$ alors $f^* > \overline{z}$.*
(ii) Soit x une variable de P et v une valeur de son domaine. Soit $P_{x=v}$ (resp. $SP(u)_{x=v}$) la restriction de P (resp. $SP(u)$) à l'ensemble des solutions telles que $x = v$, et soit $f^*_{x=v} \leq +\infty$ (resp. $f(u)_{x=v} \leq +\infty$) sa valeur minimale. Si $f(u)_{x=v} > \overline{z}$ alors $f^*_{x=v} > \overline{z}$.

Démonstration. La proposition (i) du Théorème 1 est triviale car, $SP(u)$ étant une relaxation de P, $f(u) \leq f^*$. La proposition (ii) se déduit de (i) et du fait qu'ajouter une contrainte $x = v$ à P puis appliquer une relaxation lagrangienne ou appliquer la relaxation lagrangienne à P puis ajouter la contrainte $x = v$ produit la même formulation. □

On établit la relation entre une instance de `muticost-regular`(X, Z, Π, c), avec $|Z| = R + 1$, et deux instances du RCSPP et du RCLPP comme suit : On sélectionne une variable de coût, par exemple z^0, et on crée R ressources, une par variables de coûts restantes. Le graphe considéré est $G = (\Pi_n, c^0)$. Une solution réalisable du RCSPP (resp. RCLPP) est un chemin dans Π_n depuis la source (dans la couche 0) vers un puits (dans la couche n) dont la consommation pour chaque ressource $1 \leq r \leq R$ est au moins $\underline{z^r}$ et au plus $\overline{z^r}$. De plus, nous voulons limiter par une borne supérieure $\overline{z^0}$ la valeur minimale du RCSPP (resp. une borne inférieure $\underline{z^0}$ la valeur maximale du RCLPP). Les arcs de Π_n coincident avec les variables binaires du modèle linéaire de ces deux instances.

Intéressons nous à un sous-problème lagrangien $SP(u)$ du RCSPP (l'approche est symétrique pour le RCLPP). Une légère modification de l'algorithme de `cost-regular` permet de résoudre $SP(u)$, mais aussi de filtrer des arcs de Π_n selon le Théorème 1 et de mettre à jour la borne inférieure $\underline{z^0}$. L'algorithme considère d'abord $c^0(u)$, défini par l'équation (5), comme valuation du graphe Π_n. Puis il calcule pour chaque noeud (i, q), le plus court chemin $\underline{k_{iq}^-}$ depuis la couche 0 et le plus court chemin $\underline{k_{iq}^+}$ vers la couche n. On obtient la valeur optimale $f(u) = \underline{k_{0s}^+} + \kappa_u$. Comme il s'agit d'une borne inférieure pour z^0, on peut éventuellement mettre à jour $\underline{z^0} = \max\{f(u), \underline{z^0}\}$. Les arcs sont ensuite à nouveau parcourus, afin de filter tout arc $((i-1, q), a, (i, q')) \in \delta_i$ tel que $\underline{k_{(i-1)q}^-} + c^0(u)_{ia} + \underline{k_{iq'}^+} > \overline{z^0} - \kappa_u$.

L'algorithme de filtrage complet de `multicost-regular` procède comme suit : u est initialisé à 0, l'algorithme du sous-gradient guide le choix des sous-problèmes lagrangiens auxquels est appliqué l'algorithme de filtrage décrit ci-dessus. Dans nos expérimentations, le nombre d'itérations de l'algorithme du sous-gradient est limité à 20 (il termine le plus souvent en 5 ou 6 itérations). On résout d'abord le problème de minimisation (RCSPP) puis celui de maximisation (RCLPP). Enfin, l'algorithme original de `cost-regular` est utilisé sur chaque variable de coûts de manière indépendante afin de réduire leurs bornes (Il peut également filtrer certains arcs, même si cela n'est jamais arrivé au cours de nos tests).

4 Modéliser des problèmes de planification de personnel

Dans cette section, nous montrons comment modéliser les principales règles métiers qui apparaissent dans les problèmes de planification de personnel (Personnel Scheduling Problem ou PSP) à l'aide d'une seule instance de `multicost-regular`. Nous démontrons ainsi la simplicité d'utilisation d'un tel modèle en automatisant la modélisation de PSP.

4.1 Réglementations standards

Dans un PSP, il existe plusieurs types de règles métiers. Ces dernières peuvent être regroupées en catégories : les motifs rationnels, les contraintes de cardinalité et les contraintes glissantes.

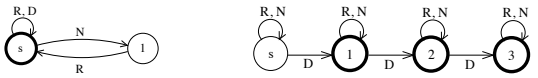
Afin d'illustrer ces catégories, considérons une planification sur 7 jours de 3 types d'activités à affecter par jour : nuit (N), jour (D) ou repos (R). Une planification s'écrit par exemple :

R	R	D	D	N	R	D

Les motifs rationnels sont modélisables directement par un AFD. Par exemple, la règle « une période de nuit est suivie par un repos » se représente par l'automate 4 (A). Une règle peut être définie aussi bien par un motif interdit ou obligatoire. Dans le premier cas, il suffira de construire l'automate complémentaire.

Les règles de cardinalité permettent de borner le nombre d'occurrences d'une ou plusieurs activités pendant une période donnée. De telles règles peuvent se modéliser à l'aide d'un automate ou d'un compteur. L'AFD de la Figure 4 (B) représente la règle « au moins 1 et au plus 3 journées chaque semaine ». Cependant, avec cette formulation, on s'aperçoit que l'état initial a été dupliqué 3 fois afin de représenter le nombre maximum de transitions D acceptables. Cela peut devenir problématique lorsque le nombre maximum d'occurrences augmente. Dans ce cas, il est plus approprié d'utiliser un compteur, ainsi il suffit de créer une nouvelle variable de coûts $z^r \in [1, 3]$ avec $c^r_{ia} = 1$ si $a = D$ et $c^r_{ia} = 0$ sinon. Plus généralement, il se peut que l'on veuille contraindre la cardinalité d'un motif. On peut également le gérer à l'aide d'un coût en isolant le motif dans l'automate décrivant les emplois du temps valides, puis d'ajouter un coût de 1 à la transition menant à ce motif.

Les contraintes glissantes peuvent également être modélisées par un AFD grâce à la reformulation proposée par Bessière et al. [2]. Cependant, la séquence glissante ne doit pas être trop longue car la reformulation impose de décrire tous les n-uplets réalisables de la contrainte glissante.



(A) Exemple de motif rationnel (B) Exemple de règle de cardinalité.

FIG. 4 – Exemples de différentes règles métiers sous forme d'automates.

4.2 Génération systématique de `multicost-regular`

Un formalisme basé sur un schéma XML et permettant de décrire des instances de PSP a été proposée dans [8]. Nous avons développé un framework capable d'interpréter ces fichiers XML afin de générer automatiquement un modèle PPC basé sur `multicost-regular`. Tout d'abord nous associons une catégorie de règles métier à chaque balise XML. Ainsi, nous pouvons pour chaque règle générer soit l'automate soit le compteur correspondant selon la catégorie à laquelle elle appartient. Par exemple, le motif interdit « pas de travail de jour après une nuit » sera défini dans le fichier XML ainsi :

```
<Pattern weight="1350"><Shift>N</Shift>
<Shift>D</Shift></Pattern>
```

Il sera automatiquement traduit en une expression rationnelle équivalente $(D|N|R) * ND(D|N|R)*$. Nous utilisons une librairie java spécialisée pour la manipulation des automates[5] afin de créer les AFD à partir des expressions rationnelles et de pouvoir par la suite les manipuler. Les fonctions complément, intersection et minimisation sont utilisées pour produire un AFD unique. Une fois cet automate construit, les règles qui produisent des compteurs sont traitées. Une instance de `multicost-regular` est générée pour chaque employé. Les dernières contraintes intégrées au modèle CSP sont les contraintes transversales. Par exemple, les demandes d'activités pour chaque jour sont représentées par une contrainte globale de cardinalité `gcc`. Notons que nous ne sommes pas capables de gérer les coûts de violation d'une règle dans `multicost-regular`. De plus, nous ne savons pas transformer de manière automatique un AFD afin d'intégrer des règles de cardinalité sur des motifs.

4.3 Deux cas de planification de personnel

Nous avons d'abord étudié un problème appelé *GPost* [8]. Ce PSP consiste à construire un emploi du

[5] http ://www.brics.dk/automaton/

temps valide de 28 jours pour 8 personnes. Chaque jour, un employé doit être affecté à une activité de jour (D), de nuit (N) ou de repos (R). Chaque employé est lié à un contrat (temps plein ou partiel) défini par un ensemble de règles. Les règles de motifs rationnels identifiées sont : « Un repos doit durer au moins deux jours », « Les WE consécutifs travaillés sont limités » et « Certaines successions de périodes sont interdites ». En utilisant notre méthode de modélisation automatique, nous construisons un AFD pour chaque type de contrat. Les contraintes de cardinalités identifiées : « Au plus n jours travaillés par mois », « Le nombre d'occurences de certains types d'activité est limité » et « Le nombre de jours travaillés par semaine est limité ». Les besoins d'activités journalières ainsi que les disponibilités des employés sont aussi définies et modélisées. Le coût de violation des règles a été ignoré, de même que la première règle de motif afin d'éviter l'irréalisabilité du problème.

Notre deuxième étude porte sur l'ensemble des problèmes de référence générés par Demassey et al. [4]. Les règles métiers sont issues d'un problème de planification de personnel réel. L'objectif est ici de construire un emploi du temps constitué de 96 périodes de 15 minutes pour une seule personne. On fait correspondre à chaque période soit une activité travaillée, une pause, un repas ou du repos. Chaque affectation possède son propre coût. On cherche à trouver l'emploi du temps de coût minimum respectant toutes les règles métiers, des motifs rationnels : « Une activité dure au moins une heure », « Changer d'activité nécessite une pause ou un repas », « Pause, repas et repos ne peuvent être consécutifs », « Les repos doivent être placés en début et en fin de journée », et « une pause dure 15 minutes ». ainsi que des contraintes de cardinalité : « Au moins 1 et au plus 2 pauses par jour », « Au moins un repas par jour » et « Entre 3 et 8 heures d'activité par jour ». À ces règles métiers s'ajoute l'interdiction de faire certains couples période-activité. Des contraintes unaires permettent de les modéliser.

5 Expérimentations

Nos expérimentations ont été effectuées sur Intel Core 2 Duo 2Ghz avec 2048Mo de RAM sous OS X. Les deux PSP présentés ont été résolus en utilisant la librairie de contraintes en java *CHOCO*. L'heuristique de sélection de valeurs par défaut, *min value*, a été appliquée.

5.1 À propos de la taille de l'automate

Comme vu dans la section 2.1, la complexité de l'algorithme de filtrage de `regular` dépend de la taille du graphe déployé, égale dans le pire des cas au produit du nombre de variables par la taille de l'automate. Si un tel algorithme peut sembler inapplicable en théorie sur des automates de trop grandes tailles, nos résultats expérimentaux mettent deux choses en évidence. Tout d'abord, les opérations utilisées pour construire automatiquement un AFD à partir de plusieurs règles tendent à générer un automate déjà partiellement déployé (suite aux intersections) et à réduire le nombre d'états redondants dans les différentes couches (suite à la minimisation). Grâce à cela, le graphe déployé généré pendant la première phase d'initialisation de la contrainte peut être largement plus petit que l'automate passé en paramètre Π. Aussi, la seconde phase d'initialisation réduit considérablement encore le graphe déployé Π_n car de nombreux états finaux ne peuvent pas être atteints en exactement n transitions. Le Tableau 1 donne le nombre de noeuds et d'arcs dans les différents automates construits lors de la construction d'une contrainte `multicost-regular` pour le problème *GPost* avec, dans l'ordre : la somme des tailles des AFD générés pour chacune des règles, la taille de l'AFD Π après intersection et minimisation, l'AFD déployé après les deux phases d'initialisation de la contrainte (phase Avant, puis phase Arrière Π_n).

Contract	Count	$\sum$AFD	Π	Avant	Π_n
Fulltime	# Nodes	5782	682	411	230
	# Arcs	40402	4768	1191	400
Parttime	# Nodes	4401	385	791	421
	# Arcs	30729	2689	2280	681

TAB. 1 – Illustration de la réduction des graphes avant résolution.

5.2 Comparaisons expérimentales

Dans la section précédente, nous avons vu la facilité de modélisation offerte par `multicost-regular`. Néanmoins, il n'y aurait pas d'intérêt à définir une telle contrainte si la rapidité de résolution était impactée. Nous avons donc comparé notre algorithme avec un modèle décomposé, liant une contrainte `regular` (ou `cost-regular` pour l'optimisation) à une contrainte `global-cardinality` (`gcc`).

Les résultats expérimentaux de l'instance *GPost* sont présentés dans le Tableau 2. Le modèle pour ce problème est composé de 8 `multicost-regular` (ou 8 `regular` et `gcc`) – une par personne – et de 28 `gcc` transversales – une par période. Deux variantes de l'instance sont considérées : la première (resp. seconde) ligne du tableau correspond au problème sans (resp. avec) la contrainte glissante limitant le nombre de week-ends consécutifs travaillés. Nous avons testé plusieurs heuristiques de choix de variables. Choisir les

	multicost-regular		regular ∧ gcc	
WE ?	Time (s)	# Fails	Time (s)	# Fails
no	1.94	24	12.6	68035
yes	16.0	1576	449.2	2867381

TAB. 2 – Résultats du problème *GPost*

variables correspondant à une même période d'abord donne les meilleurs résultats. Cette heuristique permet au solveur de contraintes de traiter plus efficacement les gcc transversales. Les deux modèles mènent aux mêmes solutions. En fait, le temps moyen passé pour chaque noeud est beaucoup plus important avec multicost-regular. Cependant grâce à un filtrage plus efficace, la taille de l'arbre de recherche et le temps global de résolution pour trouver une solution réalisable sont considérablement réduits.

Notre deuxième jeux de tests évalue le comportement de la contrainte multicost-regular (MCR) par rapport à cost-regular∧gcc (CR) lorsque l'on augmente la taille du graphe. Les tests sont effectués sur le problème d'optimisation défini dans la Section 4.3. Les modèles ne contiennent pas d'autre contrainte, cependant, le modèle décomposé CR nécessite l'utilisation de variables de channeling supplémentaires. L'ensemble de tests présenté dans le Tableau 3 compte 110 instances. Le nombre n d'activités varie entre 1 et 50. La taille de l'automate ($4n+7$ états et $10n+5$ transitions) varie ainsi de 11 à 207 états et de 15 à 505 transitions. Les coûts d'affectation ont été générés aléatoirement. Plusieurs heuristiques de choix de variables ont été testées et la meilleure pour chaque modèle a été retenue. Les résultats du modèle CR sont plus sensibles au choix de l'heuristique que le modèle MCR.

Le première partie du Tableau 3 montre que le modèle MCR permet de résoudre toutes les instances (Colonne #) en moins de 15 secondes pour les plus difficiles (Colonne t). Le nombre moyen de backtracks (Colonne bt) reste stable et bas lorsque n augmente. Le modèle CR est en revanche beaucoup plus sensible à l'augmentation de n comme le montre la seconde partie du tableau. En effet, la taille du graphe augmentant, celui-ci contient de plus en plus de chemins violant les contraintes de cardinalité. Ces chemins ne sont pas supprimés par le filtrage résultant du modèle décomposé CR. En tout, 40 instances de plus de 8 activités n'ont pas été résolues en moins de 30 minutes (Colonne #). Sur ces instances, l'optimum est atteint (mais non prouvé) dans seulement 4 cas et la déviation moyenne de la meilleure solution trouvée à l'optimum est de $4,7\%$. Si on ne s'intéresse qu'aux instances résolues, le temps de résolution (t) et le nombre de backtracks (bt) sont toujours beaucoup plus élevés avec le modèle décomposé CR.

	MCR model			CR model		
n	#	t	bt	#	t	bt
1	10	0.6	49	10	1.1	292
2	10	0.8	54	10	2.4	539
4	10	1.5	65	10	13.5	1638
6	10	1.6	44	10	53.6	4283
8	10	2.1	51	9	209.2	5132
10	10	2.4	58	7	283.5	6965
15	10	3.8	59	6	283.9	4026
20	10	4.9	49	6	311.8	4135
30	10	6.9	51	1	313.0	4303
40	10	13.4	68	0	-	-
50	10	14.4	51	1	486.0	1406

TAB. 3 – Résultats des problèmes d'optimisation

6 Conclusion

Dans ce papier, nous introduisons la contrainte globale multicost-regular et proposons une implémentation simple d'un filtrage basé sur une relaxation lagrangienne. Les expérimentations sur des problèmes de planification de personnel montrent son efficacité et son extensibilité par rapport aux modèles décomposés utilisés habituellement pour décrire des règles métiers. De plus, nous explorons une méthode systématique pour construire des instances de multicost-regular à partir d'un ensemble de règles métiers. Nous chercherons par la suite à créer un outil automatisé lié au solveur de contraintes *CHOCO* capable de modéliser et de résoudre une grande variété de problèmes d'emploi du temps.

Remerciements

Nous remercions Mats Carlsson pour avoir fourni l'exemple illustrant le défaut de propagation de l'algorithme de cost-regular. Nous remercions également Christian Schulte pour ses commentaires avisés sur le papier et ses conseils pour améliorer la contrainte.

Références

[1] N. Beldiceanu, M. Carlsson, R. Debruyne, and T. Petit. Reformulation of Global Constraints Based on Constraint Checkers. *Constraints*, 10(3), 2005.

[2] Christian Bessière, Emmanuel Hebrard, Brahim Hnich, Zeynep Kiziltan, Claude-Guy Quimper, and Toby Walsh. Reformulating global constraints : The SLIDE and REGULAR constraints. In *SARA*, pages 80–92, 2007.

[3] S. Boyd, L. Xiao, and A. Mutapcic. Subgradient methods. *lecture notes of EE392o, Stanford University, Autumn Quarter*, 2004, 2003.

[4] Sophie Demassey, Gilles Pesant, and Louis-Martin Rousseau. A cost-regular based hybrid column generation approach. *Constraints*, 11(4) :315–333, 2006.

[5] G. Handler and I. Zang. A dual algorithm for the restricted shortest path problem. *Networks*, 10 :293–310, 1980.

[6] Serdar Kadioglu and Meinolf Sellmann. Efficient context-free grammar constraints. In *AAAI*, pages 310–316, 2008.

[7] M. Maher, N. Narodytska, C.-G. Quimper, and T. Walsh. Flow-Based Propagators for the *sequence* and Related Global Constraints. In *Proceedings of CP'2008*, volume 5202 of *LNCS*, pages 159–174, 2008.

[8] Personnel Scheduling Data Sets and Benchmarks. `http ://www.cs.nott.ac.uk/ tec/NRP/`.

[9] Gilles Pesant. A regular language membership constraint for finite sequences of variables. In *Proceedings of CP'2004*, pages 482–495, 2004.

[10] Jean-Charles Régin and Jean-Francois Puget. A filtering algorithm for global sequencing constraints. In *CP*, pages 32–46, 1997.

[11] Meinolf Sellmann. Theoretical foundations of CP-based lagrangian relaxation. *Principles and Practice of Constraint Programming –CP 2004*, pages 634–647, 2004.

[12] NZ Shor, KC Kiwiel, and A Ruszcayǹski. Minimization methods for non-differentiable functions. *Springer-Verlag New York, Inc.*, 1985.

[13] M. Trick. A dynamic programming approach for consistency and propagation for knapsack constraints, 2001.

[14] Willem Jan van Hoeve, Gilles Pesant, and Louis-Martin Rousseau. On global warming : Flow-based soft global constraints. *J. Heuristics*, 12(4-5) :347–373, 2006.

Fusion de réseaux de contraintes qualitatives par morceaux

Jean-François Condotta, Souhila Kaci, Pierre Marquis et Nicolas Schwind

Université Lille-Nord de France, Artois, F-62307 Lens
CRIL, F-62307 Lens
CNRS UMR 8188, F-62307 Lens
{condotta,kaci,marquis,schwind}@cril.fr

Résumé

La représentation et le raisonnement sur le temps et l'espace est une problématique importante pour de nombreuses applications de l'intelligence artificielle : la compréhension du langage naturel, la navigation de robots, etc. Ces trentes dernières années de nombreux formalismes dits qualitatifs ont été proposés dans le but de représenter un ensemble d'entités spatiales ou temporelles et leur relations. Nous nous intéressons dans cet article au problème de la fusion de réseaux de contraintes qualitatives (RCQ en abrégé). Nous décrivons un algorithme de fusion qui construit un RCQ représentant une vision globale de l'ensemble des RCQ, éventuellement conflictuels, donnés en entrée. Cet algorithme est générique car il est indépendant du formalisme qualitatif dans lequel sont définis les RCQ. En comparaison avec une précédente approche pour la fusion de RCQ, la méthode que nous proposons est plus efficace puisqu'elle est basée sur une fusion locale des contraintes portant sur les mêmes variables (sans considérer dans un premier temps leurs interactions avec les autres contraintes). Nous définissons plusieurs opérateurs de fusion de contraintes de telle sorte que le RCQ construit résultant possède certaines propriétés d'un point de vue logique. Nous montrons en outre comment rendre le processus de fusion traitable en imposant certaines restrictions.

Abstract

Representing space and time and reasoning about them are important issues for many AI applications. In the past three decades numerous qualitative formalisms have been proposed for this purpose. In this paper the problem of merging qualitative constraints networks (QCNs) is addressed. We point out a merging algorithm which computes a QCN representing a global view of the input set of (possibly conflicting) QCNs. This algorithm is generic in the sense that it does not depend on a specific qualitative formalism. Compared to previous approaches to QCNs merging, the efficiency of our method comes from the fact that it first merges locally the constraints of the input QCNs bearing on the same pairs of variables (without considering in this first step their interactions with the other constraints). We define several constraint merging operators in a way to ensure that the induced QCNs merging operator satisfies some expected properties from a logical standpoint. We also point out some restrictions which make the corresponding QCNs merging operators tractable.

1 Introduction

La représentation et le raisonnement sur des entités spatiales ou temporelles représente une tâche importante dans de nombreux domaines de l'Intelligence Artificielle. [2, 8, 7]. Ces 30 dernières années de nombreux formalismes qualitatifs ont été développés pour représenter un ensemble d'entités spatiales ou temporelles et leur relations. Dans le domaine spatial, les relations utilisées peuvent être de type topologiques [15] (lorsque les entités considérées représentent des régions de points) ou basées sur une relation d'ordre [12]. Dans le domaine temporel, les relations d'Allen [1] permettent de représenter différentes relations de précédence entre des intervalles de la droite des rationnels. Plus récemment des formalismes basés sur les relations d'Allen ont été proposés [4, 3], ainsi que des combinaisons de formalismes qualitatifs [6].

La plupart de ces formalismes utilisent des réseaux de contraintes qualitatives (RCQ) pour représenter un ensemble d'entités spatiales ou temporelles et leurs positions relatives.

Dans certaines applications, en particulier de type multi-agents, l'information spatiale ou temporelle provient de différentes sources, c'est-à-dire que chaque

source fournit un RCQ représentant ses connaissances sur l'ensemble des configurations d'un même ensemble d'entités. En raison de la multiplicité des sources, les RCQ fournis sont généralement conflictuels. Il faut alors mettre en œuvre un processus de fusion pour résoudre les conflits. Par exemple, considérons l'exemple suivant. Trois cours communs (Bases de données, Prolog et Algorithmique) sont prévus pour un groupe d'étudiants. Ces derniers ont la possiblité d'exprimer leur préférences sur la planification de ces cours. Supposons qu'un premier étudiant veuille suivre Bases de données avant Prolog, et Prolog avant Algorithmique, et qu'un autre étudiant préfère acquérir des notions d'Algorithmique avant de suivre le cours Bases de données. Clairement aucun planning ne peut satisfaire pleinement tous les étudiants, et nous devons trouver un compromis cohérent pour résoudre les conflits.

Ces quinze dernières années de nombreux opérateurs de fusion ont été définis et étudiés dans le cadre de la logique propositionnelle [16, 9]. Une adaptation directe de certains de ces travaux pour la fusion de RCQ a été proposée dans [5]. Elle consiste à définir un opérateur de fusion qui prend en entrée un ensemble fini de RCQ représentant l'information fournie par les différentes sources, et retourne comme résultat un ensemble cohérent d'informations spatiales ou temporelles représentant une vision globale de ces RCQ. En revanche il n'existe pas d'implémentation pratique de cette approche en raison sa complexité importante.

Dans cet article nous présentons une méthode efficace pour la fusion de RCQ. Cette nouvelle méthode consiste à fusionner localement les contraintes des RCQ portant sur les mêmes paires de variables, en exploitant les opérateurs de fusion propositionnelle basés sur des calculs de distance [9]. Nous définissons une procédure de fusion de RCQ de manière à obtenir comme résultat de la fusion un RCQ *cohérent*.

La suite de cet article est organisée comme suit : le paragraphe suivant présente les définitions et notations utilisées à propos des formalismes spatio-temporels, et en particulier à propos des RCQ. Dans le paragraphe 3 nous introduisons au travers d'un exemple le problème de la fusion d'un ensemble de RCQ conflictuels. Nous décrivons dans le paragraphe 4 notre nouvelle méthode de fusion de RCQ, exploitant un opérateur de fusion local sur les contraintes basé sur un calcul de distances. Dans le paragraphe 5, nous montrons comment améliorer l'algorithme de fusion en imposant certaines restrictions. Enfin nous concluons dans le dernier paragraphe en présentant quelques perspectives pour des travaux futurs.

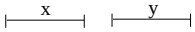
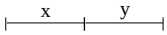
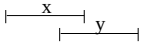
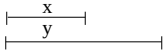
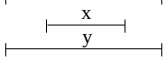
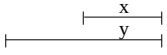
Relation	Symbole	Inverse	Illustration
x before y	b	bi	
x meets y	m	mi	
x overlaps y	o	oi	
x starts y	s	si	
x during y	d	di	
x finishes y	f	fi	
x equal y	eq	eq	

FIGURE 1 – L'ensemble $\mathcal{B}_{int}$ des 13 relations de base de l'algèbre des intervalles.

2 Rappels sur les algèbres qualitatives et les RCQ

Une algèbre qualitative considère un ensemble fini $\mathcal{B}$ de relations binaires appelées *relations de base* sur un domaine $\mathcal{D}$. Chacune des relations de base de $\mathcal{B}$ représente une situation qualitative particulière entre deux entités prenant leur valeurs dans $\mathcal{D}$. Ces relations sont supposées complètes et mutuellement exclusives, autrement dit deux éléments de $\mathcal{D}$ satisfont une et une seule relation de base de $\mathcal{B}$. $2^{\mathcal{B}}$ dénote l'ensemble de tous les sous-ensembles de $\mathcal{B}$. Les éléments de $2^{\mathcal{B}}$ sont appelés *relations*. Une relation $R \in 2^{\mathcal{B}}$ représente l'ensemble de relations de base possibles entre deux éléments de $\mathcal{D}$. L'une des relations de base est la relation identité, notée *eq*. Chacune des relations de base r de $\mathcal{B}$ est associée à une relation inverse $r^{-1} \in \mathcal{B}$ telle que $\forall X, Y \in \mathcal{D}, XrY$ ssi $Yr^{-1}X$.

Par exemple, la figure 1 décrit l'ensemble $\mathcal{B}_{int}$ des 13 relations de base de l'algèbre d'Allen (ou algèbre des intervalles) pouvant être satisfaites entre deux intervalles de la droite des rationnels. Nous avons $\mathcal{B}_{int} = \{eq, b, bi, m, mi, o, oi, s, si, d, di, f, fi\}$.

Un réseau de contraintes qualitatives (RCQ en abrégé) est utilisé pour représenter un ensemble de configurations qualitatives possibles entre des entités. Etant donné un ensemble de relations de base $\mathcal{B}$, un RCQ N défini sur $2^{\mathcal{B}}$ est une paire (V, C) où $V = \{v_1, \ldots, v_n\}$ est un ensemble de variables représentant les entités spatiales ou temporelles considérées, et C est une application qui associe à chaque paire de variables (v_i, v_j) une relation de $2^{\mathcal{B}}$. $C(v_i, v_j)$ est également appelée *contrainte*. Nous préférerons la notation C_{ij} plutôt que $C(v_i, v_j)$ par souci de concision. Pour tout $v_i, v_j \in V$, $C_{ji} = \{r \mid r^{-1} \in C_{ij}\}$ et $C_{ii} = \{eq\}$.

Nous introduisons maintenant quelques définitions sur les RCQ. Soit $\mathcal{B}$ un ensemble de relations de base

et $N = (V, C)$ un RCQ défini sur $2^{\mathcal{B}}$. Une *instanciation cohérente* de N sur $V' \subseteq V$ est une application α de V' dans $\mathcal{D}$ telle que $\alpha(v_i)\ C_{ij}\ \alpha(v_j)$, pour tout $v_i, v_j \in V'$. N est dit *cohérent* ssi il existe une instanciation cohérente de N sur V. Un *sous-réseau* de N est un RCQ $N' = (V, C')$ où $C'_{ij} \subseteq C_{ij}$ pour tout $v_i, v_j \in V$. Un *scénario cohérent* de N est un sous-réseau cohérent de N dans lequel chaque contrainte est composée d'une et une seule relation de base de $\mathcal{B}$ (une telle contrainte est alors appelée contrainte atomique).

$[N]$ représente l'ensemble des scénarios cohérents de N. Nous notons N^V_{ALL} le RCQ (V, C) tel que pour tout $v_i, v_j \in V$, $C_{ij} = \mathcal{B}$ si $v_i \neq v_j$ et $C_{ij} = \{eq\}$ sinon.

3 Fusion de RCQ

Etant donné un ensemble de relations de base $\mathcal{B}$, nous considérons un ensemble $\mathcal{N} = \{N_1, \ldots, N_m\}$ de RCQ définis sur $2^{\mathcal{B}}$ et sur un ensemble commun de variables $V = \{v_1, \ldots, v_n\}$.

Exemple 1. *Considérons les trois RCQ définis sur $2^{\mathcal{B}_{int}}$ et sur $V = \{v_1, v_2, v_3, v_4\}$, schématisés sur la figure 2. Nous utlisons les conventions suivantes pour la représentation schématique d'un RCQ : pour tout v_i, v_j, nous ne représentons ni les contraintes $C_{ii} = \{eq\}$, ni les contraintes C_{ji} si C_{ij} est représentée, puisque $C_{ji} = \{r \mid r^{-1} \in C_{ij}\}$. Enfin lorsqu'un RCQ ne fournit aucune information entre deux variables (par exemple, la paire (v_2, v_3) pour le RCQ N_1), la contrainte associée correspond implicitement à l'ensemble $\mathcal{B}$ et elle n'est pas représentée.*

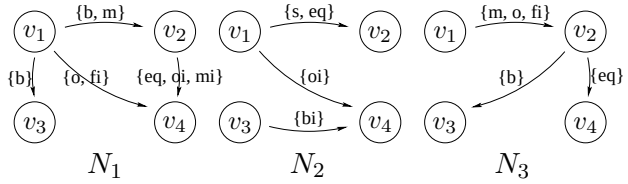


FIGURE 2 – Trois RCQ à fusionner.

Notre but est d'obtenir un ensemble cohérent d'informations représentant de manière globale l'ensemble des RCQ de $\mathcal{N}$. La manière la plus naturelle de répondre à ce but est de prendre comme résultat l'ensemble de tous les scénarios cohérents admis par chacun des RCQ de $\mathcal{N}$, c'est-à-dire l'ensemble $\bigcap_{N_i \in \mathcal{N}} [N_i]$. Cependant cet ensemble peut être vide. En effet, considérons l'exemple ci-dessus. Chacun des scénarios appartenant à l'ensemble $[N_1]$ admet une des relations de base de $\{b, m\}$ entre les variables v_1 et v_2, et chacun des scénarios de $[N_2]$ admet une des relations de base de $\{s, eq\}$ entre ces mêmes variables. Il n'existe pas de scénario cohérent commun à $[N_1]$ et $[N_2]$, et donc $\bigcap_{N_i \in \mathcal{N}} [N_i] = \emptyset$.

Condotta et al. [5] ont proposé une méthode parcimonieuse pour la fusion de RCQ, qui retourne en résultat un ensemble non vide de scénarios cohérents les plus proches (en termes de distance) de l'ensemble $\mathcal{N}$. Dans cette approche, l'algorithme de fusion nécessite le calcul de distances à propos de tous les scénarios cohérents possibles sur V, donc appartenant à l'ensemble $[N^V_{ALL}]$. En outre l'ensemble de scénarios cohérents résultants peut être de taille exponentielle. Ces deux critères rendent cette approche difficile à mettre en œuvre.

Nous décrivons dans cet article une approche pratique pour la fusion de RCQ. Cette approche fusionne indépendamment les contraintes portant sur les mêmes variables. Notre méthode est basée sur l'algorithme suivant. Pour chaque paire de variables (v_i, v_j) nous appliquons un opérateur de fusion sur l'ensemble de toutes les contraintes des RCQ entre v_i et v_j. Nous obtenons alors une nouvelle contrainte correspondant à l'ensemble des relations de base de $\mathcal{B}$ représentant une information globale pour la contrainte entre v_i et v_j. Nous obtenons un nouvel ensemble de contraintes C, chacune représentant une vision globale des contraintes correspondantes de chaque RCQ à fusionner. Nous définissons alors le résultat de la fusion comme étant le RCQ $N_{res} = (V, C)$.

4 Un nouvel algorithme pour la fusion de RCQ

4.1 Fusion de contraintes

Rappelons que notre but est de fusionner un ensemble de RCQ $\mathcal{N} = \{N_1, \ldots, N_m\}$ définis sur $2^{\mathcal{B}}$ et sur le même ensemble de variables V. L'approche consiste à fusionner indépendamment les contraintes pour chaque paire de variables (v_i, v_j) de V en utilisant un opérateur de fusion ϑ pour les relations de $2^{\mathcal{B}}$. Par conséquent notre approche nécessite la définition d'un opérateur de fusion de contraintes ϑ.

Définition 1 (opérateur de fusion de contraintes). *Soit $\mu \in 2^{\mathcal{B}}$ une relation appelée* relation d'intégrité. *ϑ est un opérateur qui associe à une relation d'intégrité μ et un multi-ensemble de relations de $2^{\mathcal{B}}$ noté $\mathcal{R}$, une relation de $2^{\mathcal{B}}$ notée R. ϑ est un* opérateur de fusion de contraintes *s'il satisfait au moins les propriétés suivantes, $\forall \mu \in 2^{\mathcal{B}}, \forall \mathcal{R}$ ensemble de relations de $2^{\mathcal{B}}$:*

(a) $\vartheta(\mu, \mathcal{R}) \subseteq \mu$,
(b) si $\mu \neq \emptyset$, *alors* $\vartheta(\mu, \mathcal{R}) \neq \emptyset$,
(c) si $\mu \cap \bigcap_{R \in \mathcal{R}} R \neq \emptyset$, *alors* $\vartheta(\mu, \mathcal{R}) = \mu \cap \bigcap_{R \in \mathcal{R}} R$.

La propriété *(a)* assure que le résultat de la fusion est une relation dont les relations de base appartiennent à la relation d'intégrité μ. La propriété *(b)* garantit que si μ n'est pas la relation vide alors le résultat de la fusion n'est pas la relation vide. La propriété *(c)* exige que $\vartheta(\mu, \mathcal{R})$ soit l'ensemble des relations de base appartenant à μ et à chacune des relations de $\mathcal{R}$, si cet ensemble n'est pas vide.

Soit $\mathcal{N}$ un ensemble de RCQ définis sur $2^{\mathcal{B}}$ et V. Notons $V^2_<$ l'ensemble $\{(v_i, v_j) \in V \times V \mid i < j\}$. Pour chaque paire de variables (v_i, v_j) de $V^2_<$, nous désignons par $\mathcal{C}_{ij}$ l'ensemble des contraintes des RCQ de $\mathcal{N}$ portant sur la paire de variable (v_i, v_j). L'algorithme 1 décrit la construction par morceaux du RCQ résultat $N_{res} = (V, C)$.

Algorithme 1 : Fusion par morceaux d'un ensemble de RCQ

Entrée : un ensemble $\mathcal{N}$ de RCQ définis sur $2^{\mathcal{B}}$ et V,
un opérateur de fusion de contraintes ϑ

1 **début**
2 $\quad N_{res} = (V, C)$;
3 $\quad \mu = \mathcal{B}$;
4 $\quad$ **pour tout** $(v_i, v_j) \in V^2_<$ **faire**
5 $\quad\quad C_{ij} = \vartheta(\mu, \mathcal{C}_{ij})$;
6 $\quad$ **fin faire**
7 $\quad$ **retourner** N_{res};
8 **fin**

Nous pouvons observer que pour tout ensemble $\mathcal{C}_{ij}$ à fusionner nous n'imposons aucune restriction sur μ, le sur-ensemble du résultat devant être renvoyé par l'opérateur ϑ (cf. ligne 3 de l'algorithme). Ceci est dû au fait que nous ne considérons pas les dépendances entre les contraintes résultantes du RCQ N_{res} pour l'instant.

Notons que puisque nous obtenons en résultat un RCQ, la complexité spatiale de cette méthode, linéaire en la taille de l'entrée, est beaucoup plus intéressante que celle proposée dans [5], dans laquelle le résultat renvoyé est de taille exponentielle. Sa complexité temporelle dépend toutefois de la construction de ϑ.

Pour définir un opérateur de fusion de contraintes ϑ au sens de la définition 1, nous exploitons les travaux effectués dans [9] pour la fusion d'opérateurs de fusion de bases propositionnelles basés sur un calcul de distances. Le principal ingrédient utilisé dans de tels opérateurs de fusion est une « distance », ce qui permet d'obtenir comme résultat un ensemble de connaissances les plus « proches » de l'ensemble à fusionner. Dans le cadre de la logique propositionnelle, étant donné un ensemble $\mathcal{K}$ de bases de croyances propositionnelles à fusionner et une formule propositionnelle IC (pour *contrainte d'intégrité*), le résultat de la fusion correspond à l'ensemble des modèles de IC qui sont les plus proches (en termes de distance) de l'ensemble $\mathcal{K}$.

Dans le même esprit, étant donné un ensemble $\mathcal{R} = \{R_1, \ldots, R_p\}$ de relations de $2^{\mathcal{B}}$ et une relation $\mu \subseteq \mathcal{B}$ représentant une relation d'intégrité, on peut définir un opérateur de fusion de contraintes en trois étapes.

Tout d'abord, on calcule une distance locale entre chacune des relations de base de μ et chaque relation $R_i \in \mathcal{R}, i \in \{1, \ldots, p\}$. La distance entre une relation de base $r_1 \in \mu$ et une relation R est la distance minimale entre r_1 et chacune des relations de base de R. Formellement,

$$d(r_1, R) = \begin{cases} \min\{d_r(r_1, r_2) \mid r_2 \in R\} & \text{si } R \neq \emptyset \\ 0 & \text{sinon.} \end{cases}$$

Nous devons donc disposer d'une distance locale entre relations de base de $\mathcal{B}$.

Définition 2 (**distance entre relations de base**). *Une distance d_r entre deux relations de base de $\mathcal{B}$ est une pseudo-distance, c'est-à-dire une application de $\mathcal{B} \times \mathcal{B}$ dans $\mathbb{R}^+$ telle que $\forall r_1, r_2 \in \mathcal{B}$,*

$$\begin{cases} d_r(r_1, r_2) = d_r(r_2, r_1) \\ d_r(r_1, r_2) = 0 \text{ } ssi \text{ } r_1 = r_2 \\ d_r(r_1, r_2) = d_r(r_1^{-1}, r_2^{-1}), \end{cases}$$

où r_1^{-1} (resp. r_2^{-1}) est la relation inverse de r_1 (resp. r_2).

Par exemple, pour la distance drastique, la distance entre deux relations de base est égale à 1 si elles sont différentes, 0 sinon.

Dans le but de définir une distance locale d_r plus pertinente, il est important de rappeler qu'une algèbre qualitative incorpore un concept de voisinage entre relations de base. Ce voisinage est usuellement représenté par un treillis sur l'ensemble des relations de base de $\mathcal{B}$. Pour de nombreux formalismes qualitatifs un tel treillis existe [11, 12]. Ce treillis permet alors de déterminer un graphe de voisinage conceptuel entre relations de base. La figure 3 schématise le graphe de voisinage entre les relations de base de $\mathcal{B}_{int}$, correspondant au diagramme de Hasse du treillis des intervalles [11]. La distance de voisinage conceptuel entre relations de base [5] exploite ce graphe et est définie comme suit :

Définition 3 (**distance de voisinage conceptuel**). *Soit $\mathcal{G}_{\mathcal{B}}$ le diagramme de Hasse du treillis sur $\mathcal{B}$. La distance de voisinage conceptuel entre deux relations de base r et r', notée $d_v(r, r')$ est la longueur de la chaîne la plus courte dans $\mathcal{G}_{\mathcal{B}}$ reliant r à r'.*

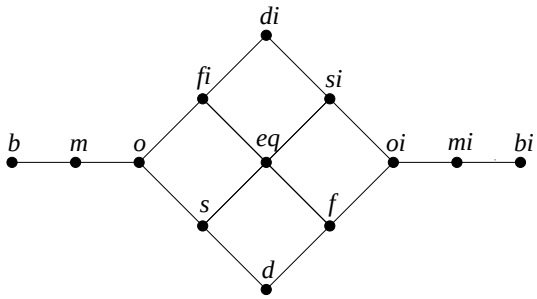


FIGURE 3 – Le graphe de voisinage conceptuel de l'algèbre des intervalles.

Par exemple, pour des relations de base de $\mathcal{B}_{int}$, on obtient $d_v(di, f) = 3$ puisque (di, fi, eq, f) est l'une des chaînes les plus courtes entre di et f dans le graphe de voisinage associé (cf. Figure 3).

La deuxième étape consiste à agréger les distances locales calculées à l'étape précédente afin d'obtenir une distance globale entre une relation de base r et l'ensemble $\mathcal{R}$, en utilisant une fonction d'agrégation.

Définition 4 (**fonction d'agrégation**). *Une fonction d'agrégation est une application $\otimes$ qui essocie à un ensemble de nombres réels positifs un nombre réel positif, et qui satisfait les propriétés suivantes :*

- *si $x_1 \leq x'_1, \ldots, x_n \leq x'_n$, alors $\otimes(x_1, \ldots, x_n) \leq \otimes(x'_1, \ldots, x'_n)$ (monotonie)*
- *si $x_1 = \ldots = x_n = 0$, alors $\otimes(x_1, \ldots, x_n) = 0$ (minimalité)*

Nous considérons une propriété supplémentaire pour une fonction d'agrégation $\otimes$:

Définition 5 (**minimalité forte**). *Une fonction d'agrégation $\otimes$ satisfait le postulat de minimalité forte ssi*

$$si \ \otimes(x_1, \ldots, x_n) = 0, \ alors \ x_1 = \ldots = x_n = 0.$$

Plusieurs fonctions d'agrégation ont été étudiés dans la littérature [13, 16]. Par exemple, la fonction de majorité $\sum$ [13] qui effectue la somme de ses arguments favorise le point de vue de la majorité des sources. La fonction d'arbitrage $\mathcal{MAX}$ [16] qui retourne quant à elle l'argument de valeur maximale sera plus appropriée si l'on désire minimiser l'insatisfaction maximale sur l'ensemble des sources par rapport au résultat. Etant donné une fonction d'agrégation, on calcule la distance globale $d_\otimes$ entre une relation de base r et l'ensemble $\mathcal{R}$. Cette distance globale est définie comme suit :

$$d_\otimes(r, \mathcal{R}) = \otimes\{d(r, R) \mid R \in \mathcal{R}\}.$$

Enfin, le résultat de la fusion, noté $\vartheta^{d_r,\otimes}(\mu, \mathcal{R})$, est le sous-ensemble des relations de base de μ les plus « proches » de l'ensemble $\mathcal{R}$. Formellement,

$$\vartheta^{d_r,\otimes}(\mu, \mathcal{R}) = \{r \in \mu \mid \nexists r' \in \mu : d_\otimes(r', \mathcal{R}) < d_\otimes(r, \mathcal{R})\}.$$

Proposition 1. *Si $\otimes$ est une fonction d'agrégation vérifiant le postulat de minimalité forte, alors $\vartheta^{d_r,\otimes}$ est un opérateur de fusion de contraintes, au sens de la définition 1.*

Nous donnons les preuves de quelques propositions en annexe.

Remarquons que les fonctions d'agrégation $\sum$ et $\mathcal{MAX}$ satisfont le postulat de minimalité forte. Il en résulte que $\vartheta^{d_r,\sum}$ et $\vartheta^{d_r,\mathcal{MAX}}$ sont des opérateurs de fusion de contraintes, au sens de la définition 1.

Exemple 1 (**suite**). *Considérons de nouveau les trois RCQ de l'exemple (cf. Figure 2). Soit $\mathcal{C}_{12} = \{\{b, m\}, \{s, eq\}, \{m, o, fi\}\}$ l'ensemble des contraintes des RCQ portant sur la paire de variables (v_1, v_2), et soit $\mu = \mathcal{B}$. Pour chaque relation de base $r \in \mu$ on calcule une distance locale entre r et chaque relation de $\mathcal{C}_{12}$. Par exemple, en utilisant la distance de voisinage conceptuel entre relations de base, la distance locale entre la relation de base b et la relation $\{m, o, fi\}$ est donnée par*

$$\begin{aligned} d(b, \{m, o, fi\}) &= \min\{d_v(b, m), d_v(b, o), d_v(b, fi)\} \\ &= \min\{1, 2, 3\} = 1. \end{aligned}$$

En utilisant la fonction d'agrégation $\otimes = \sum$, on obtient

$$d_\sum(d(b, \{b, m\}), d(b, \{s, eq\}), d(b, \{m, o, fi\})) = 4.$$

Le résultat de la fusion $\vartheta^{d_v,\sum}(\mu, \mathcal{C}_{12})$ correspond aux relations de base de μ les plus proches de l'ensemble des relations de $\mathcal{C}_{12}$, c'est-à-dire l'ensemble $\{m, o\}$.

Evaluons maintenant la complexité de calcul de $\vartheta^{d_r,\otimes}(\mu, \mathcal{R})$. Notons $|E|$ le nombre d'éléments d'un ensemble fini E. $f(\otimes, |\mathcal{R}|)$, respectivement $f(d_r)$ correspond au nombre maximum d'étapes du calcul de $\otimes$ sur ses $|\mathcal{R}|$ arguments, repectivement d_r. Comme d_r est une distance entre relations de base d'un ensemble donné $\mathcal{B}$, on a $f(d_r) \in O(1)$. La distance d entre une relation de base et une relation se calcule alors également en temps constant, puisque $|\mathcal{R}|$ est borné par une constante. Donc le calcul de $\vartheta^{d_r,\otimes}(\mu, \mathcal{R})$ s'effectue en $\alpha \cdot f(\otimes, |\mathcal{R}|)$ étapes de calcul pour une certaine constante α. En outre, la plupart des fonctions d'agrégation « usuelles » sont calculables en temps polynomial, et donc dans ce cas la fusion de contraintes s'effectue en temps polynomial. Par exemple, si $\otimes \in \{\sum, \mathcal{MAX}\}$, on a $f(\otimes, |\mathcal{R}|) \in O(|\mathcal{R}|)$. Dans ce cas $\vartheta^{d_r,\otimes}(\mu, \mathcal{R})$ est calculable en temps $O(|\mathcal{R}|)$.

Ces résultats se détachent considérablement de la fusion de bases propositionnelles qui est typiquement un problème NP-difficile ; par contre des résultats de complexité similaires pourraient être obtenus en limitant le nombre de variables propositionnelles à une certaine constante donnée.

4.2 Maintenir la cohérence du RCQ résultant

Nous n'avons aucune garantie à propos de la cohérence du RCQ résultant de la fusion N_{res}. Dans le but de résoudre ce problème, notre approche est basée sur l'idée suivante : nous construisons l'ensemble des contraintes résultantes C étape par étape. A l'initialisation du processus, chacune des contraintes de C est définie par l'ensemble $\mathcal{B}$ (autrement dit N_{res} est initialisé au RCQ N^V_{ALL}). A chaque étape, nous choisissons une paire (v_i, v_j) dans $V^2_<$, et nous associons à la contrainte C_{ij} le résultat de la fusion de l'ensemble $\mathcal{C}_{ij}$ tout en préservant la cohérence du RCQ.

Pour cela, nous supposons l'existence d'un ordre total $\prec$ sur $V^2_<$ dans le but de résoudre le problème du choix de la contrainte à chaque étape : les paires de variables sont alors sélectionnées les unes après les autres en respectant l'ordre $\prec$.

Cet ordre peut être obtenu de différentes manières. Par exemple toutes les paires de variables de $V^2_<$ peuvent être ordonnées en fonction d'un certain degré de priorité ou de fiabilité, lui-même résultant de l'agrégation de degrés locaux définis sur chaque RCQ. Par défaut, cet ordre est défini comme étant l'ordre lexicographique sur les indices des variables.

Le processus global de fusion est donné dans l'algorithme 2.

L'algorithme initialise tout d'abord chaque contrainte du RCQ résultant N_{res} à la relation universelle $\mathcal{B}$ (lignes 3 à 5). Puis il sélectionne la prochaine paire (v_i, v_j) de variables de $V^2_<$ en respectant l'ordre $\prec$ (cette paire correspond à la prochaine contrainte C_{ij} à affecter, ligne 6). Des lignes 7 à 12, SAT est un test de cohérence du RCQ donné en paramètre. Notons $N[(i,j,r)]$ le RCQ $N = (V, C)$ dans lequel on assigne à la contrainte C_{ij} la relation de base r. On définit la relation d'intégrité μ_{ij} comme étant l'ensemble des relations de base $r \in \mathcal{B}$ pour lesquelles $N_{res}[(i,j,r)]$ est cohérent. Ensuite, la contrainte courante C_{ij} est définie comme étant le résultat de la fusion de $\mathcal{C}_{ij}$ sous la relation d'intégrité μ_{ij}, ligne 13. L'algorithme se termine lorsque toutes les contraintes de N_{res} ont été assignées.

Dès lors que le problème SAT est un problème NP-difficile, même si ϑ est calculable en temps polynomial, l'algorithme 2 ne calcule pas le RCQ résultant en temps polynomial. En effet il effectue un nombre d'appels linéaire à SAT (linéaire en nombre de contraintes $|V^2_<|$ à traiter).

Algorithme 2 : Fusion cohérente d'un ensemble de RCQ

Entrée : un ensemble $\mathcal{N} = \{N_1, \ldots, N_m\}$ de RCQ sur V,
un opérateur de fusion de contraintes ϑ,
un ordre total $\prec$ sur l'ensemble $V^2_<$
Sortie : un RCQ $N_{res} = (V, C)$

```
1  début
2      N_res = (V, C);
3      pour tout (v_i, v_j) ∈ V²_< faire
4          C_ij = B;
5      fin faire
6      pour tout (v_i, v_j) ∈ V²_< dans l'ordre ≺ faire
7          μ_ij = ∅;
8          pour tout r ∈ B faire
9              si SAT(N_res[(i, j, r)]) alors
10                 μ_ij = μ_ij ∪ {r};
11             finsi
12         fin faire
13         C_ij = ϑ(μ_ij, C_ij);
14     fin faire
15     retourner N_res;
16 fin
```

L'algorithme 2 donne lieu à la proposition suivante :

Proposition 2. *N_{res} est cohérent.*

La proposition suivante exprime le fait que fusionner localement les contraintes des RCQ conduit à une propriété intéressante à propos du RCQ final : si les RCQ donnés en entrée ne sont pas conflictuels, alors l'algorithme 2 retourne en résultat un RCQ dont l'ensemble des scénarios cohérents correspond exactement à l'ensemble des scénarios cohérents communs à chacun des RCQ donnés en entrée.

Proposition 3.

$$\bigcap_{N_i \in \mathcal{N}} [N_i] \neq \emptyset \Rightarrow [N_{res}] = \bigcap_{N_i \in \mathcal{N}} [N_i].$$

Exemple 1 (suite). *Nous appliquons l'algorithme 2 pour fusionner l'ensemble des trois RCQ $\mathcal{N} = \{N_1, N_2, N_3\}$ de notre même exemple (cf. Figure 2). Pour $\vartheta^{d_r,\otimes}$ nous choisissons $d_r = d_v$, la distance de voisinage conceptuel entre relations de base, et $\otimes = \sum$. Nous définissons l'ordre $\prec$ comme étant l'ordre lexicographique sur les indices des variables. La figure 4 résume le processus complet qui consiste à affecter étape par étape chaque contrainte du RCQ résultant N_{res} en respectant l'ordre $\prec$. N_{res} est le RCQ cohérent obtenu après l'étape 6. Chaque contrainte de N_{res} est composée des relations de base les plus proches de l'ensemble*

des contraintes correspondantes dans les RCQ de $\mathcal{N}$ de telle sorte qu'à chaque étape N_{res} est maintenu cohérent.

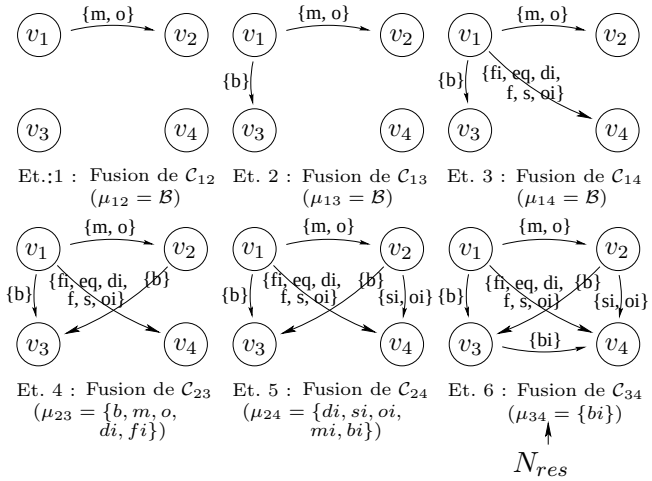


FIGURE 4 – Le processus de fusion, étape par étape.

5 Amélioration de l'agorithme

5.1 Eviter des tests de cohérence inutiles

Afin d'affecter chacune des contraintes de N_{res}, l'algorithme 2 effectue à chaque fois un nombre constant d'appels à la fonction SAT (lignes 7 à 12). Notre but est maintenant d'améliorer l'algorithme en supprimant les appels inutiles à SAT pour certaines contraintes.

Dans cette section, nous considérerons les algèbres qualitatives vérifiant la propriété suivante : soit deux RCQ cohérents $N_1 = (V_1, C_1)$ et $N_2 = (V_2, C_2)$ définis sur $2^{\mathcal{B}}$, et respectivement définis sur deux ensembles disjoints de variables, et soit le RCQ N défini sur $V = V_1 \cup V_2$ tel que N_1 et N_2 en sont des sous-réseaux. Pour tout $v_i \in V_1$, $v_j \in V_2$, $r \in \mathcal{B}$, le RCQ $N[(i, j, r)]$ conserve sa cohérence. Cette propriété est vérifiée pour la plupart des algèbres qualitatives définies dans la littérature, en particulier toutes celles dont les relations considérées sont basées sur une relation d'ordre [1, 3, 11, 12]. A notre connaissance, cette propriété reste indémontrée dans le cas des relations de type topologiques [15].

Etant donné un RCQ $N = (V, C)$ sur $2^{\mathcal{B}}$, nous dirons que C_{ij} est *inactive* si $C_{ij} = \mathcal{B}$, *active* sinon.

Proposition 4. *Soit $N = (V, C)$ un RCQ cohérent sur $\mathcal{B}$ et (v_i, v_j) une paire de variables de V. S'il n'existe pas de chaîne de contraintes actives de N $C_{ik}, \ldots, C_{pj}$, alors $\forall r \in \mathcal{B}$, $N[(i, j, r)]$ est cohérent.*

Supposons que pendant l'exécution de l'algorithme 2, la contrainte à affecter est C_{ij}. D'après la proposition 4 nous pouvons remplacer les lignes 7 à 12 par les suivantes :

si *il n'existe pas de chaîne de contraintes actives dans N_{res} entre v_i et v_j* **alors**
  $\mu_{ij} = \mathcal{B}$;
sinon
  `// lignes 7 à 12 de l'algorithme 2`
finsi

Soit $\mathcal{N}$ un ensemble de RCQ définis sur V. Si pour une paire (v_i, v_j) de V toutes les contraintes de $\mathcal{C}_{ij}$ sont définies par l'ensemble $\mathcal{B}$ (autrement dit, aucune des sources n'a d'information à propos de la contrainte entre les variables v_i et v_j), alors nous dirons qu'une telle paire est *non contrainte par rapport à $\mathcal{N}$*, *contrainte par rapport à $\mathcal{N}$* sinon.

Définition 6 (**arbre-ensemble**). *Soit $\mathcal{N}$ un ensemble de RCQ définis sur V. $\mathcal{N}$ est un arbre-ensemble s'il n'existe pas de cycle de paires de variables de V contraintes par rapport à $\mathcal{N}$.*

Proposition 5. *Soit $\mathcal{N}$ un ensemble de RCQ définis sur V, ϑ un opérateur de fusion de contraintes, et $\prec$ un ordre quelconque sur $V^2_<$. Si $\mathcal{N}$ est un arbre-ensemble, alors pour toute paire (v_i, v_j) de V contrainte par rapport à $\mathcal{N}$, $\forall r \in \mathcal{B}$, $N_{res}[(i, j, r)]$ est cohérent.*

Par conséquent, si $\mathcal{N}$ est un arbre-ensemble de RCQ, les lignes 7 à 12 de l'algorithme 2 peuvent alors simplement être remplacées par l'instruction $\mu_{ij} = \mathcal{B}$. Nous pouvons alors en déduire le lemme suivant :

Lemme 1. *Si $\mathcal{N}$ est un arbre-ensemble, alors l'algorithme 2 s'exécute en temps polynomial. De plus, le RCQ résultant N_{res} est indépendant de l'ordre $\prec$ sur $V^2_<$ choisi.*

5.2 Exploitation des classes traitables

En général le problème de la cohérence d'un RCQ est un problème NP-difficile. Dans le but de pallier ce problème de nombreux fragments traitables ont été identifiés dans différentes algèbres qualitatives [14, 12]. Etant donnée une algèbre qualitative définie sur l'ensemble de relations de base $\mathcal{B}$, considérons un sous-ensemble $\mathcal{S}$ de $2^{\mathcal{B}}$. $\mathcal{S}$ est qualifié de sous-ensemble traitable ssi la cohérence de tout RCQ dont les contraintes sont définies par un élément de $\mathcal{S}$ peut être décidée en temps polynomial. Dans notre cadre, $\mathcal{S}$ doit être nécessairement fermée pour l'intersection ($\cap$), définie comme l'intersection ensembliste usuelle, et l'inverse ($^{-1}$), définie pour tout $R \in 2^{\mathcal{B}}$ comme étant la relation $R^{-1} = \{r \mid r^{-1} \in R\}$. De plus nous imposons le fait que $\mathcal{S}$ contienne la relation universelle $\mathcal{B}$ et la relation atomique $\{eq\}$.

Lorsqu'un tel ensemble traitable $\mathcal{S}$ est identifié, l'algorithme 2 peut etre amélioré en assignant chacune des contraintes du RCQ construit N_{res} à une relation de $\mathcal{S}$. En procédant ainsi tous les tests de cohérence (ligne 9 de l'agorithme) peuvent être effectués en temps polynomial, et ainsi l'algorithme 2 peut s'exécuter en temps polynomial.

Dans cette optique, pour chaque paire $(v_i, v_j) \in V_<^2$, nous calculons $\vartheta^{d_r,\otimes}(\mu_{ij}, \mathcal{C}_{ij})$ et nous assignons alors la contrainte résultante C_{ij} à la relation de $\mathcal{S}$ la plus « proche » de $\vartheta^{d_r,\otimes}(\mu_{ij}, \mathcal{C}_{ij})$. Il est donc nécessaire de disposer d'une distance $d_R(R_1, R_2)$ entre deux relations de $2^{\mathcal{B}}$.

Définition 7 (**distance entre relations**). *Une distance d_R entre deux relations de $2^{\mathcal{B}}$ est une pseudo-distance, c'est-à-dire une application de $2^{\mathcal{B}} \times 2^{\mathcal{B}}$ dans $\mathbb{R}^+$ telle que $\forall R_1, R_2 \in 2^{\mathcal{B}}$,*

$$\begin{cases} d_R(R_1, R_2) = d_R(R_2, R_1) \\ d_R(R_1, R_2) = 0 \text{ ssi } R_1 = R_2 \end{cases}$$

Nous considérons une propriété supplémentaire pour une distance d_R entre relations :

Définition 8 (**inégalité stricte**). *Une distance d_R entre relations satisfait le postulat d'inégalité stricte ssi $\forall R_1, R_2, R_3 \in 2^{\mathcal{B}}$,*

$$R_1 \subset R_2 \subset R_3 \Rightarrow d_R(R_2, R_3) < d_R(R_1, R_3).$$

Dans cette section, nous imposons que la distance entre relations utilisée vérifie le postulat d'inégalité stricte. Cette restriction reste faible. En effet, plus généralement, la plupart des distances entre ensembles d'éléments quelconques satisfont ce postulat. Une distance entre relations peut être définie de différentes manières, nous rappelons l'une des plus naturelles, la distance euclidienne.

Définition 9 (**distance euclidienne entre relations**). *Soit d une distance locale entre une relation de base de $\mathcal{B}$ et une relation de $2^{\mathcal{B}}$. La distance euclidienne entre deux relations de $2^{\mathcal{B}}$ est définie comme suit, $\forall R_1, R_2 \in 2^{\mathcal{B}}$:*

$$d_{Euc}(R_1, R_2) = \sqrt{\sum_{r \in \mathcal{B}} (d(r, R_1) - d(r, R_2))^2}.$$

Soit $N_{res}^* = (V, C)$ le RCQ résultant de la fusion. Pour chaque paire $(v_i, v_j) \in V_<^2$, la contrainte C_{ij} est assignée à la relation $R_{ij}^* \in \mathcal{S}$ la plus « proche » de $\vartheta^{d_r,\otimes}(\mu_{ij}, \mathcal{C}_{ij})$ par rapport à d_R, avec $R_{ij}^* \supseteq \vartheta^{d_r,\otimes}(\mathcal{C}_{ij})$. Formellement, pour toute paire (v_i, v_j) de $V_<^2$, R_{ij}^* est la relation de $\mathcal{S}$ telle que $R_{ij}^* \supseteq \vartheta^{d_r,\otimes}(\mu_{ij}, \mathcal{C}_{ij})$, et $\nexists R \in \mathcal{S}$ telle que $R \supseteq \vartheta^{d_r,\otimes}(\mu_{ij}, \mathcal{C}_{ij}), d_R(R, \vartheta^{d_r,\otimes}(\mu_{ij}, \mathcal{C}_{ij})) < d_R(R_{ij}^*, \vartheta^{d_r,\otimes}(\mu_{ij}, \mathcal{C}_{ij}))$.

Proposition 6. *$\forall (v_i, v_j) \in V_<^2$, R_{ij}^* existe et est unique.*

Par conséquent nous remplaçons la ligne 13 de l'agorithme 2 par l'instruction $C_{ij} = R_{ij}^*$.

Puisque pour chaque paire $(v_i, v_j) \in V_<^2$ nous avons $R_{ij}^* \supseteq \vartheta^{d_r,\otimes}(\mu_{ij}, \mathcal{C}_{ij})$, chaque contrainte C_{ij} est instanciée par une relation qui contient toutes les relations de base de $\vartheta^{d_r,\otimes}(\mu_{ij}, \mathcal{C}_{ij})$. Donc d'après la proposition 2, nous avons la proposition suivante :

Proposition 7. *N_{res}^* est cohérent.*

Si les RCQ en entrée ne sont pas conflictuels, l'ensemble des scénarios cohérents de N_{res}^* ne coincide pas nécessairement avec l'ensemble des scénarios communs à chacuns des RCQ en entrée, comme c'est le cas en utilisant la méthode générale (cf. Proposition 3). Néanmoins nous avons le résultat suivant, plus faible :

Proposition 8. *Si toutes les contraintes des RCQ de $\mathcal{N}$ sont définies par des relations de $\mathcal{S}$, alors*

$$\bigcap_{N_i \in \mathcal{N}} [N_i] \neq \emptyset \Rightarrow [N_{res}^*] = \bigcap_{N_i \in \mathcal{N}} [N_i].$$

6 Conclusion

Nous avons proposé dans cet article une nouvelle méthode efficace pour la fusion de réseaux de contraintes qualitatives (RCQ) définis sur une algèbre qualitative commune. Notre méthode est générique puisque qu'elle capture la plupart des algèbres qualitatives définies dans la littérature. Etant donné un ensemble de RCQ éventuellement conflictuels, le processus de fusion retourne un RCQ cohérent représentant une vision globale des RCQ en entrée. Il consiste à fusionner localement les contraintes des RCQ. En s'inspirant des travaux effectués dans le cadre de la fusion de bases propositionnelles, nous avons défini une classe d'opérateurs de fusion de contraintes basés sur un calcul de distance qui permet de fusionner localement les contraintes des RCQ en entrée pour construire pas à pas les contraintes du RCQ résultant. Nous avons proposé un compromis intéressant entre efficacité du nouveau processus et propriétés logiques sur le RCQ résultant, et nous avons également montré comment obtenir un algorithme de fusion en temps polynomial en imposant certaines restrictions.

Ce travail peut être étendu dans plusieurs directions. La définition de l'ordre $\prec$ n'a aucune influence sur le RCQ résultant de la fusion lorsque l'ensemble de RCQ à fusionner est un arbre-ensemble de RCQ (cf. paragraphe 5). Cependant cette notion d'arbre-ensemble de RCQ peut paraître assez restrictive. Une problématique de recherche future est de caractériser de manière

plus faible la structure de l'ensemble de RCQ à fusionner et d'étudier comment on peut réduire l'influence de la définition de l'ordre $\prec$ sur le RCQ résultant.

Konieczny et al. [10] ont proposé une caractérisation logique des opérateurs de fusion propositionnelle avec contraintes d'intégrité sous la forme d'un ensemble de postulats de rationalité. Un travail futur sera d'inclure une extension de ces postulats dans le contexte des opérateur de fusion de contraintes.

Références

[1] J-F. Allen. An interval-based representation of temporal knowledge. In *IJCAI*, pages 221–226, 1981.

[2] J-F. Allen. Towards a general theory of action and time. *Artificial Intelligence*, 23(2) :123–154, 1984.

[3] P. Balbiani and J-F. Condotta. Spatial reasoning about points in a multidimensional setting. *Applied Intelligence*, 17(3) :221–238, 2002.

[4] P. Balbiani and A. Osmani. A model for reasoning about topological relations between cyclic intervals. In *KR*, pages 378–385, 2000.

[5] J-F. Condotta, S. Kaci, and N. Schwind. A framework for merging qualitative constraints networks. In *FLAIRS*, pages 586–591, 2008.

[6] A. Gerevini and J. Renz. Combining topological and size information for spatial reasoning. *Artificial Intelligence*, 137 :1–42, 2002.

[7] I. Hamlet and J. Hunter. A representation of time for medical expert systems. In *AIME*, pages 112–119, 1987.

[8] H. A. Kautz. *A formal theory of plan recognition.* PhD thesis, Rochester, NY, USA, 1987.

[9] S. Konieczny, J. Lang, and P. Marquis. Distance-based merging : a general framework and some complexity results. In *KR*, pages 97–108, 2002.

[10] S. Konieczny and R. P. Pérez. Merging with integrity constraints. In *ECSQARU*, pages 233–244, 1999.

[11] G. Ligozat. On generalized interval calculi. In *AAAI*, pages 234–240, 1991.

[12] G. Ligozat. Reasoning about cardinal directions. *Journal of Visual Languages and Computing*, 9(1) :23–44, 1998.

[13] J. Lin. Integration of weighted knowledge bases. *Artificial Intelligence*, 83 :363–378, 1996.

[14] B. Nebel. Solving hard qualitative temporal reasoning problems : Evaluating the efficiency of using the ORD-Horn class. *Constraints*, 1(3) :175–190, 1997.

[15] D-A. Randell, Z. Cui, and A. Cohn. A spatial logic based on regions and connection. In *KR*, pages 165–176. 1992.

[16] P. Z. Revesz. On the semantics of arbitration. *Journal of Algebra and Computation*, 7(2) :133–160, 1997.

Annexe : preuves de quelques propositions

Proposition 1. *Si $\otimes$ est une fonction d'agrégation vérifiant le postulat de minimalité forte, alors $\vartheta^{d_r,\otimes}$ est un opérateur de fusion de contraintes, au sens de la définition 1.*

Preuve. Soit $\otimes$ une fonction d'agrégation vérifiant le postulat de minimalité forte. La propriété *(a)* est vérifiée par définition de $\vartheta^{d_r,\otimes}$, et la propriété *(b)* est vérifiée puisque $\mathcal{B}$ est un ensemble fini. Nous prouvons la propriété *(c)* en deux étapes :
- Montrons d'abord $\mu \cap \bigcap_{R\in\mathcal{R}} R \subseteq \vartheta^{d_r,\otimes}(\mu,\mathcal{R})$. Si $\mu \cap \bigcap_{R\in\mathcal{R}} R = \emptyset$, le résultat est trivial. Soit $r \in \mu \cap \bigcap_{R\in\mathcal{R}} R$. $\forall R \in \mathcal{R}$, on obtient $d(r,R) = 0$. Donc $d_\otimes(r,\mathcal{R}) = 0$ (minimalité de $\otimes$). Donc $\forall r' \in \mu, d_\otimes(r,\mathcal{R}) \leq d_\otimes(r',\mathcal{R})$. On a alors $r \in \vartheta^{d_r,\otimes}(\mu,\mathcal{R})$.
- Montrons maintenant $\vartheta^{d_r,\otimes}(\mu,\mathcal{R}) \subseteq \mu \cap \bigcap_{R\in\mathcal{R}} R$. Soit $r \in \vartheta^{d_r,\otimes}(\mu,\mathcal{R})$. On a $r \in \mu$. Supposons $r \notin \bigcap_{R\in\mathcal{R}} R$. $\exists R_h \in \mathcal{R}$ tel que $r \notin R_h$, et donc $d(r,R_h) > 0$. On obtient $d_\otimes(r,\mathcal{R}) > 0$ puisque $\otimes$ satisfait le postulat de minimalité forte. Puisque $\mu \cap \bigcap_{R\in\mathcal{R}} R \neq \emptyset$, soit $r' \in \mu \cap \bigcap_{R\in\mathcal{R}} R$, on a $d_\otimes(r',\mathcal{R}) = 0$. Donc $\exists r' \in \mu\ d_\otimes(r',\mathcal{R}) < d_\otimes(r,\mathcal{R})$. Contradiction avec $r \in \vartheta^{d_r,\otimes}(\mu,\mathcal{R})$. □

Proposition 2. *N_{res} est cohérent.*

Preuve. Notons N^k_{res} le RCQ construit N_{res} avant l'assignation de la k^{e} constrainte en respectant l'ordre $\prec$; notons alors $\mathcal{C}^k$, respectivement C^k l'ensemble des contraintes à fusionner correspondant, respectivement la contrainte C_{ij} à affecter correspondante, notons $N[(k,r)]$ le RCQ $N = (V,C)$ dans lequel on assigne à la contrainte C^k la relation de base r, et enfin notons plutôt μ_k pour la relation d'intégrité μ_{ij} correspondante. Soit $|V^2_<|$ la taille de l'ensemble $V^2_<$, qui représente également le nombre d'assignations à effectuer au sein de l'algorithme. Montrons par récurrence que $N^{|V^2_<|}_{res}$ est cohérent.
N^1_{res} est cohérent par définition. Soit $k > 1$, et supposons N^k_{res} cohérent. On a alors $\exists r \in \mathcal{B} : N^k_{res}[(k,r)]$ est cohérent. Donc $\mu_k \neq \emptyset$. Donc d'après la propriété *(b)* de la définition 1 d'un opérateur de fusion de contraintes, on a $\vartheta(\mu_k,\mathcal{C}^k) \neq \emptyset$. Par définition de μ_k dans l'algorithme 2 (ligne 10), on a $\forall r \in \mu_k$, $N^k_{res}[(k,r)]$ est cohérent. Or d'après la propriété *(a)* de la définition 1, $\vartheta(\mu_k,\mathcal{C}^k) \subseteq \mu_k$. Donc $\forall r \in \vartheta(\mu_k,\mathcal{C}^k)$,

$N^k_{res}[(k,r)]$ est cohérent. Ce dernier RCQ correspond au RCQ N^{k+1}_{res}. Par conséquent, par récursivité N_{res} est cohérent. □

Proposition 3.

$$\bigcap_{N_i \in \mathcal{N}} [N_i] \neq \emptyset \Rightarrow [N_{res}] = \bigcap_{N_i \in \mathcal{N}} [N_i].$$

Preuve. Utilisons les notations de la preuve de la proposition précédente : N^k_{res} est le RCQ construit N_{res} avant l'assignation de la k^{e} contrainte en respectant l'ordre $\prec$; $\mathcal{C}^k$, respectivement C^k est l'ensemble des contraintes à fusionner correspondant, respectivement la contrainte C_{ij} à affecter correspondante, $N[(k,r)]$ est le RCQ $N = (V, C)$ dans lequel on assigne à la contrainte C^k la relation de base r, et μ_k est la relation d'intégrité μ_{ij}. Comme $\bigcap_{N_i \in \mathcal{N}} [N_i] \neq \emptyset$, il existe un scénario cohérent σ appartenant à $\bigcap_{N_i \in \mathcal{N}} [N_i]$. Soit σ_{ij} la relation de base contenue dans la contrainte du scénario σ entre v_i et v_j, et σ_k la relation de base de σ dans la k^{e} contrainte à affecter. Montrons par récurrence sur k que pour tout k, $\sigma_k \in \mu_k \cap \bigcap_{C' \in \mathcal{C}^k} C'$. Montrons par récurrence sur k que pour tout k, $\sigma_k \in \mu_k \cap \bigcap_{C' \in \mathcal{C}^k} C'$.

- cas de base : on a déjà $\sigma_1 \in \mu_1 \cap \bigcap_{C' \in \mathcal{C}^1} C'$. En effet puisque $\mu_1 = \mathcal{B}$, $\sigma_1 \in \mu_1$. Comme par définition de σ, $\sigma_1 \in \bigcap_{C' \in \mathcal{C}^1} C'$, on obtient bien $\sigma_1 \in \mu_1 \cap \bigcap_{C' \in \mathcal{C}^1} C'$.
- supposons que pour tout $l \in \{1, \ldots, k\}$, on ait $\sigma_l \in \mu_l \cap \bigcap_{C' \in \mathcal{C}^l} C'$. Montrons que $\sigma_{k+1} \in \mu_{k+1} \cap \bigcap_{C' \in \mathcal{C}^{k+1}} C'$. D'après l'hypothèse de récurrence, σ est un sous-réseau de $N^k[(k, \sigma_{k+1})]$, et donc $N^k[(k, \sigma_{k+1})]$ est cohérent. De la définition de μ_{k+1} on obtient $\sigma_{k+1} \in \mu_{k+1}$. Comme par définition de σ, $\sigma_{k+1} \in \bigcap_{C' \in \mathcal{C}^{k+1}} C'$, on obtient $\sigma_{k+1} \in \mu_{k+1} \cap \bigcap_{C' \in \mathcal{C}^{k+1}} C'$.

Il en résulte que pour tout k, $\mu_k \cap \bigcap_{C' \in \mathcal{C}^k} C' \neq \emptyset$. D'après la propriété *(c)* de la définition 1 d'un opérateur de fusion de contraintes, on a alors pour tout scénario σ et pour tout k, $\sigma_k \in \vartheta(\mu_k, \mathcal{C}^k)$ ssi $\sigma_k \in \mu_k \cap \bigcap_{C' \in \mathcal{C}^k} C' \neq \emptyset$. Ce qui conlut la preuve. □

Proposition 4. *Soit $N = (V, C)$ un RCQ cohérent sur $\mathcal{B}$ et (v_i, v_j) une paire de variables de V. S'il n'existe pas de chaîne de contraintes actives de N $C_{ik}, \ldots, C_{pj}$, alors $\forall r \in \mathcal{B}$, $N[(i,j,r)]$ est cohérent.*

Preuve. Soit $N = (V, C)$ un RCQ cohérent sur $2^{\mathcal{B}}$ et (v_i, v_j) une paire de variables de V. Supposons qu'il n'existe pas de chaîne de contraintes actives de N $C_{ik}, \ldots, C_{pj}$. Cela signifie que l'on peut considérer N comme deux RCQ indépendants $N_1 = (V_1, C_1)$ et $N_2 = (V_2, C_2)$ avec $v_i \in V_1$ et $v_j \in V_2$. D'après la supposition faite précédemment, C_{ij} peut être assigné à la relation $\{r\}$ pour tout $r \in \mathcal{B}$, tout en conservant la cohérence de N. □

Proposition 5. *Soit $\mathcal{N}$ un ensemble de RCQ définis sur V, ϑ un opérateur de fusion de contraintes, et $\prec$ un ordre quelconque sur $V^2_<$. Si $\mathcal{N}$ est un arbre-ensemble, alors pour toute paire (v_i, v_j) de V contrainte par rapport à $\mathcal{N}$, $\forall r \in \mathcal{B}$, $N_{res}[(i,j,r)]$ est cohérent.*

Preuve. Soit $\mathcal{N}$ un arbre-ensemble de RCQ sur V et (v_i, v_j) une paire de variables contrainte par rapport à $\mathcal{N}$. Il suffit de montrer qu'il n'existe pas de chaîne de contraintes actives de N_{res} $C_{ik}, \ldots, C_{pj}$, et la preuve découle alors de la proposition 4.
D'après la définition d'un arbre-ensemble, puisque (v_i, v_j) est une paire de variables contrainte par rapport à $\mathcal{N}$, toute chaîne $C_l = (v_i, v_{k^l}), \ldots, (v_{p^l}, v_j)$ contient une paire de variables (v_{a^l}, v_{b^l}) non contrainte par rapport à $\mathcal{N}$. Dans le graphe $\mathcal{G} = (V, A)$ où l'ensemble des arêtes A correspond à l'ensemble des paires de variables contraintes par rapport à $\mathcal{N}$, v_i et v_j appartiennent à deux composantes connexes différentes. Donc pour toute chaîne $C_l = (v_i, v_{k^l}), \ldots, (v_{p^l}, v_j)$, $\forall r \in \mathcal{B}$, $N_{res}[(a^l, b^l, r)]$ est cohérent. Donc $\mu_{a^l b^l} = \mathcal{B}$. D'autre part toute contrainte de $\mathcal{C}_{a^l b^l}$ est définie par l'ensemble $\mathcal{B}$. D'après la propriété *(c)* de la définition 1, $\vartheta(\mu_{a^l b^l}, \mathcal{C}_{a^l b^l}) = \mathcal{B}$. Donc $C_{a^l b^l} = \mathcal{B}$ est une contrainte inactive.

□

Proposition 6. *$\forall (v_i, v_j) \in V^2_<$, R^*_{ij} existe et est unique.*

Preuve. Soit $(v_i, v_j) \in V^2_<$. R^*_{ij} existe, puisque $2^{\mathcal{B}}$ est un ensemble fini et que $\mathcal{B} \in \mathcal{S}$. Prouvons l'unicité de R^*_{ij}. Si $\vartheta^{d_r,\otimes}(\mu_{ij}, \mathcal{C}_{ij}) \in \mathcal{S}$, alors $R^*_{ij} = \vartheta^{d_r,\otimes}(\mu_{ij}, \mathcal{C}_{ij})$ et la proposition est vérifiée. Sinon, soit R^1_{ij}, R^2_{ij} deux relations différentes de $\mathcal{S}$ qui s'accordent avec la définition de R^*_{ij}. Soit $R^{\cap}_{ij} = R^1_{ij} \cap R^2_{ij}$. $R^{\cap}_{ij}$ est dans $\mathcal{S}$, puisque $\mathcal{S}$ est fermé pour l'intersection. De plus, $R^{\cap}_{ij} \supseteq \vartheta^{d_r,\otimes}(\mu_{ij}, \mathcal{C}_{ij})$. Enfin pour $k \in \{1, 2\}$, $d_R(R^{\cap}_{ij}, \vartheta^{d_r,\otimes}(\mu_{ij}, \mathcal{C}_{ij})) < d_R(R^k_{ij}, \vartheta^{d_r,\otimes}(\mu_{ij}, \mathcal{C}_{ij}))$, puisque nous supposons que d_R vérifie le postulat d'inégalité stricte (cf. Définition 8). Contradiction. □

Proposition 8. *Si toutes les contraintes des RCQ de $\mathcal{N}$ sont définis par des relations de $\mathcal{S}$, alors*

$$\bigcap_{N_i \in \mathcal{N}} [N_i] \neq \emptyset \Rightarrow [N^*_{res}] = \bigcap_{N_i \in \mathcal{N}} [N_i].$$

Preuve. Puisque $\mathcal{S}$ est fermé pour l'intersection, nous avons $\forall (v_i, v_j) \in V^2_<$, $R^*_{ij} = \vartheta^{d_r,\otimes}(\mu_{ij}, \mathcal{C}_{ij})$. La preuve est alors similaire à celle de la proposition 3. □

Génération rapide de scénarios géophysiques par satisfaction de contraintes pour la localisation des séismes

Jean-Philippe Poli[1] **David Mercier**[1] **Anthony Larue**[1]
Carole Maillard[2] **Jocelyn Guilbert**[2]

[1]CEA, LIST, Laboratoire Intelligence Multicapteurs et Apprentissage
F-91191 Gif-sur-Yvette, France
[2]CEA, DAM, DIF, F-91297 Arpajon, France.

{prenom.nom}@cea.fr

Résumé

Le but du travail que nous menons est d'améliorer un système déjà existant dont le but est la localisation automatique des séismes. La localisation d'un événement à l'aide de ce système se déroule en trois temps. Dans un premier temps, les différents signaux collectés par les stations sismiques sont utilisés pour détecter les différentes heures d'arrivées des phases sismiques. Ces phases sont ensuite classées en cinq classes en fonction de la distribution temps-fréquence de leur signal. Enfin, la position de l'événement sismique est déduite de cet étiquetage des phases. Malheureusement, les heures d'arrivées des phases, appelées *pointés*, peuvent être imprécises ce qui a pour effet de perturber l'étiquetage des phases. Ces erreurs affectent la localisation du séisme du fait qu'elles mènent à des scénarios géophysiquement impossibles.

Afin d'améliorer le système existant, nous proposons d'insérer une phase supplémentaire qui consiste à déterminer l'étiquetage des phases le plus vraisemblable possible à partir des sorties des classifieurs. L'idée sous-jacente est d'étiqueter chacune des phases en considérant toutes les autres phases plutôt que d'étiqueter chacune des phases indépendamment les unes des autres, afin de s'assurer de la cohérence de l'ensemble des étiquettes. Pour cela, nous utilisons plusieurs solveurs de contraintes afin de prédire tous les étiquetages géophysiquement possibles de l'ensemble du réseau de stations. La principale difficulté d'une telle approche est en fait la taille de l'espace de recherche. En effet, pour un séisme moyen, le pointeur automatique fournit 4 pointés pour une vingtaine de stations. Puisque chaque pointé appartient à une des 5 classes, le nombre de scénarios à examiner est de l'ordre de $5^{4\times 20} \approx 8.3 \times 10^{55}$. Le cadre des CSP va nous permettre de diminuer considérablement cet espace de recherche en ne tenant plus compte des scénarios géophysiquement impossibles en réduisant sa taille à 10^{40} environs.

Dans cet article, nous décrivons la méthode retenue en insistant sur une contrainte globale qui est proche de la subsumption en logique et notre choix de contraintes nous permettant de générer une représentation condensée de l'ensemble des solutions en 25 secondes dans le pire des cas.

Abstract

The goal of our work is to improve an existing automatic location system for seismic surveillance. The present system is a three-step workflow. Firstly, it uses the signal from several stations which collect the waves of the event in order to determine the arrival times of the different phases. It then classifies the different phases into five classes, depending on the time-frequency distribution of their signal. Finally, it deduces the position of the seismic event from these labelled phases. Unfortunately, the automatic arrival times may be inaccurate and the classifier may misclassify some phases. All these mistakes strongly affect the location of the event because they can lead to impossible geophysical scenarios.

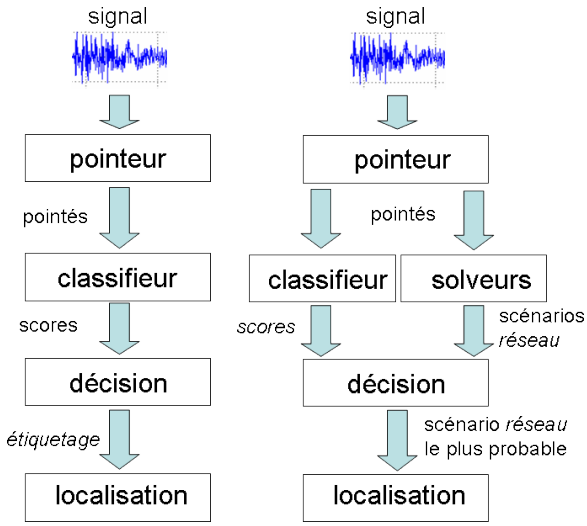
In order to improve the current system, we propose to insert another step which consists in determining the most probable phase labelling from the classifier outputs. This idea basically consists in considering the labelling of all the phases of the network at the same time instead of considering each phase individually, in order to ensure the geophysical consistency of the overall labelling. The main difficulty is the huge number of scenarios which

must be examined : for an average seismic event, the automatic picker can find 4 arrival-times for 20 stations. Each arrival-time matches with a phase which can take 5 values. The number of scenarios to be examined is thus $5^{4\times 20} \approx 8.3 \times 10^{55}$. However, constraint programming helps us to fastly generate geophysically coherent scenarios. We have split the problem into several models which work on different variable scales.

We describe in this article a subsumption-like global constraint and argue the choices of constraints which allow to browse such search-spaces in 25 seconds in the worst encountered case.

1 Introduction

1.1 Contexte



(a) Processus de localisation actuel

(b) Notre proposition

FIG. 1 – Comparaison du processus de localisation automatique et de notre proposition.

Le DASE (Département d'analyse et de surveillance de l'environnement) du CEA (Commissariat à l'Energie Atomique) est en charge des alertes sismiques en France. Dans le cadre de leur mission, les agents du DASE procèdent à une automatisation de certaines tâches qu'il est nécessaire d'accomplir avant de pouvoir localiser un séisme. Afin de rester au niveau de l'état-de-l'art et d'améliorer le système automatique en termes de précision et de rapidité, le DASE procède régulièrement à une mise à jour de certaines parties du système.

Dans [5], les auteurs proposent une première amélioration de ce système de localisation automatique avec un nouveau processus en trois phases, représenté par la figure 1(a). Dans un premier temps, le système détermine les pointés, c'est-à-dire l'horodatage des phases.

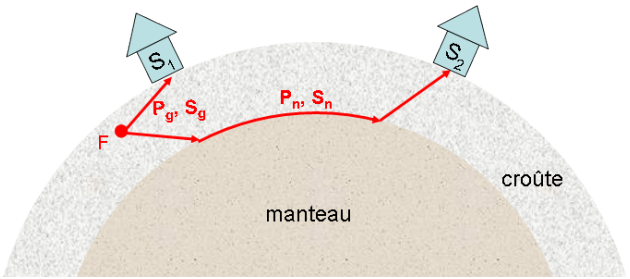


FIG. 2 – Comparaison des ondes de type n et de type g. S_1 et S_2 représentent deux stations et F le foyer du séisme.

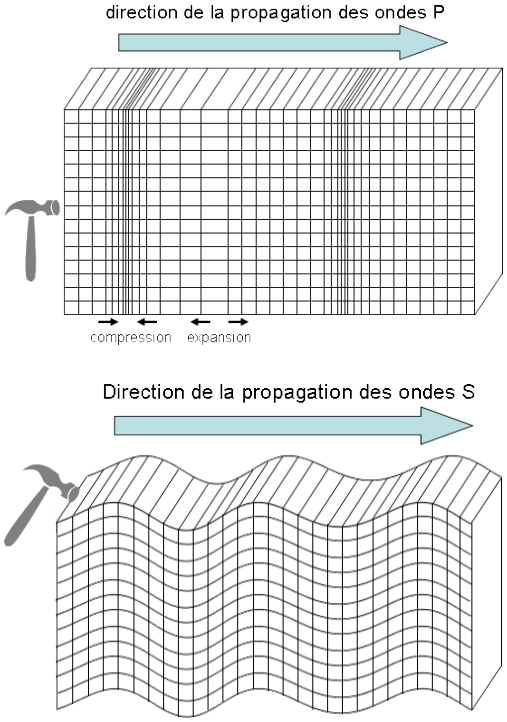


FIG. 3 – Comparaison de la propagation des ondes P et S.

En France, plus d'une quarantaine de stations ont pour but de collecter ces différentes phases. Un pointeur automatique est utilisé pour déterminer à partir de ces signaux les différents pointés qui caractérisent un séisme. Des classifieurs automatiques utilisent la portion de signal autour de ces pointés afin de leur attribuer un type de phase.

Le système proposé distingue cinq types de phases : Pg, Pn, Sg, Sn. Une classe *rejet* est aussi considérée afin de prendre en compte les erreurs éventuelles du pointeur automatique. Les quatre premières classes sont définies en fonction du mode de propagation de l'onde sismique (P ou S) et du chemin emprunté par celle-ci (n ou g), comme illustré dans la figure 3. Les

phases g sont des ondes directes tandis que les phases n sont réfléchies par l'interface croûte-manteau (figure 2). Les classifieurs fournissent ainsi un score pour chacune des classes envisagées. La décision finale consiste en un vote majoritaire pour chacune des phases. L'étiquetage des pointés d'une station est appelé un scénario *station* et l'étiquetage des pointés de l'ensemble du réseau de stations est appelé scénario *réseau*. Finalement, le scénario *réseau* ainsi obtenu permet de procéder à la localisation de l'événement. Il apparaît nettement que la précision de la localisation dépend fortement de l'exactitude de l'étiquetage des phases et de la précision du pointeur automatique, c'est-à-dire de l'exactitude des scénarios *station* et des scénarios *réseau*.

Ce système de localisation automatique peut mener à des scénarios *station* ou des scénarios *réseau* géophysiquement impossibles car aucune propriété géophysique n'est utilisée pendant la prise décision. De plus, le système souffre du fait que les phases sont étiquetées les unes indépendamment des autres. Nous proposons dans cet article une solution à ce problème.

1.2 Position du problème

Le but de notre travail est de sélectionner le scénario *réseau* le plus vraisemblable parmi l'ensemble de scénarios *réseau* possibles. Pour cela, nous souhaitons estimer la probabilité des différents scénarios en considérant les sorties des classifieurs comme des probabilités. Si l'on suppose que toutes les phases sont indépendantes les unes des autres, la probabilité d'un scénario *station* peut être estimée par le produit des probabilités des phases. Si l'on suppose aussi que toutes les stations sont indépendantes, alors la probabilité d'un scénario *réseau* peut être estimée par le produit des probabilités des scénarios *station* qui le composent. Etant données ces hypothèses, la sélection du scénario *réseau* le plus vraisemblable et géophysiquement cohérent peut être effectué en calculant les probabilités de tous les scénarios *réseau* corrects (figure 1(b)). Contrairement au processus d'étiquetage précédent, l'étiquetage d'une phase se fait à présent respectueusement de toutes les autres phases : la cohérence de l'étiquetage est dans un premier temps vérifié au niveau des stations, puis au niveau du réseau global. Dans la mesure où nous nous insérons dans un système opérationnel existant, le temps de calcul doit être une préoccupation importante et ne doit pas excéder la demi-minute.

Afin de résoudre le problème de génération des scénarios *réseau* géophysiquement possibles, nous avons choisi d'adopter la programmation par contraintes [7]. Dans un autre cadre, des réseaux de contraintes sont souvent utilisés pour reconnaître des scénarios décrits par des contraintes temporelles, comme celles proposées par Allen [1]. Par exemple, dans le domaine du multimédia, les solveurs permettent de reconnaître le genre d'un document audiovisuel [6, 2] en utilisant des contraintes pour décrire un modèle pour chaque genre considéré. Dans ces cas là, il s'agit de tester une solution potentielle pour déterminer si elle satisfait l'ensemble des contraintes, ce qui se résout en un temps polynômial.

Dans notre cas, la tâche est un peu différente : nous avons besoin d'obtenir l'ensemble des scénarios *réseau* géophysiquement possibles parmi l'ensemble des scénarios pouvant être élaborés à partir des différents pointés. Prenons le cas par exemple d'un séisme moyen : les différentes ondes issues de ce séisme sont collectées par une vingtaine de stations. En moyenne, le pointeur automatique trouve 4 pointés par station. Chacun de ces pointés peut prendre une valeur parmi les cinq considérées : Pg, Pn, Sg, Sn ou *rejet*. Il y a donc $5^4 = 625$ scénarios *station* possibles par station. Puisque une vingtaine de stations a perçu les ondes du séisme, le nombre de scénarios *réseau* est de l'ordre de $625^{20} \approx 8.3 \times 10^{55}$. Cette estimation est très pessimiste dans la mesure où les contraintes sur les scénarios *station* sont très fortes. Evidemment, la difficulté de notre travail va consister à parcourir cet espace de recherche très grand en quelques secondes. Les contraintes géophysiques, obtenues auprès des experts, ne suffisent pas à elles seules à satisfaire cette contrainte opérationnelle.

Dans la suite de cet article nous décrivons notre système composé de plusieurs modèles. Nous décrivons ensuite les contraintes utilisées et nous introduisons une nouvelle contrainte globale, comparable à la subsumption logique, que nous avons mise en place afin d'accélérer le parcours de l'espace de recherche. Nous terminerons par l'exhibition de quelques résultats expérimentaux avant de conclure.

2 Description du système

Nous avons choisi d'utiliser plusieurs modèles afin de générer la liste des scénarios *réseau* géophysiquement possibles. Ce choix a été appuyé par le fait que les différents modèles allaient utiliser des variables de types très différents. La figure 4 donne un aperçu du système.

Le premier modèle est en charge de générer les scénarios *station* possibles. Les variables à ce niveau représentent les pointés perçus par une même station par un tuple (t, l) où t est la date d'arrivée et l une étiquette parmi Pg, Pn, Sg, Sn et *rejet*. En effet, puisque les scénarios *station* ont au plus quatre phases alors que le pointeur automatique peut en trouver jusqu'à

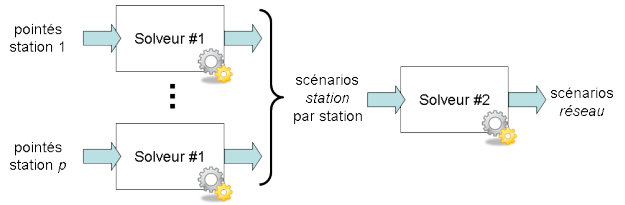


FIG. 4 – Architecture du système.

huit, la classe *rejet* nous permettra de prendre en compte des pointés issus des erreurs du pointeur automatique. Puisque t est connu pour chaque pointé, le rôle du premier modèle est d'attribuer une étiquette à chacun d'eux. Ainsi, les solutions obtenues après résolution du modèle représentent des scénarios *station* dont la cohérence est vérifiée à l'échelle de la station. Dans un soucis de performance, plusieurs instances du premier solveur sont exécutées en parallèle (une par station).

Le second modèle est composé de variables qui représentent un scnéario *station* : il y a donc autant de variables que de station ayant perçue l'événement. Ce modèle a donc pour but d'unifier les scénarios *station* en un unique scénario *réseau*. Son rôle est donc de s'assurer que l'étiquetage des phases est cohérent au niveau du réseau global. La particularité du second modèle est que les domaines des variables sont les solutions du premier modèle.

Pour chacun de ces deux niveaux de cohérences, des contraintes différentes seront utilisées. Nous décrivons ces contraintes dans la section suivante.

3 Contraintes géophysiques et parcours de l'espace de recherche

Nous avons implémenté notre propre solveur de contraintes basé sur un algorithme de *backtracking* avec filtrage [4] afin de parcourir l'espace de recherche : il s'agit d'un solveur dédié uniquement à ce problème. Si le premier modèle utilise majoritairement des contraintes globales, nous avons préféré limiter le second à des contraintes binaires ou unaires : en effet, les contraintes portent ainsi sur au plus deux stations et les instanciations partielles peuvent être plus rapidement évincées. En particulier, à chaque fois qu'une station sera ajoutée à l'instanciation partielle d'un scénario *réseau*, il suffira de la comparer à chacune des stations déjà instanciées.

Les contraintes que nous avons utilisées découlent de la principale information géophysique que nous avons, utilisé par les experts eux-mêmes : l'hodochrone (figure 5). L'hodochrone est une représentation graphique qui fournit le temps de propagation des phases en fonction

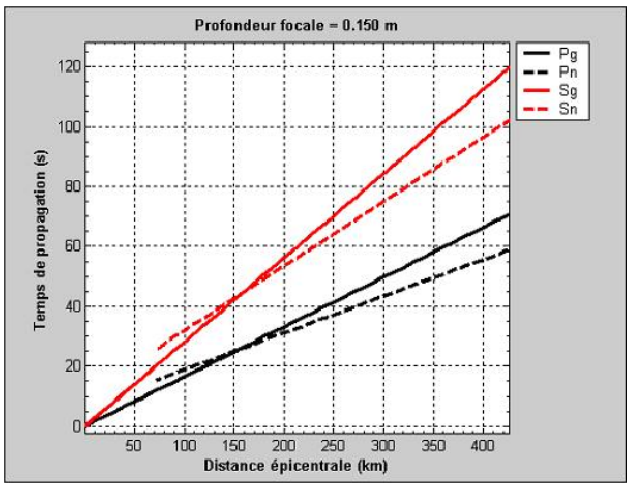


FIG. 5 – Hodochrone de référence.

de la distance entre l'événement et la station. L'hodochrone que nous utilisons n'est qu'une simplification des hodochrones réels : il s'agit d'un hodochrone de référence qui a été estimé empiriquement sur l'ensemble des séismes d'une année en France. Afin de servir de modèle, il a été simplifié. Nous supposons que les autres paramètres d'un hodochrone réel, comme par exemple la profondeur du foyer sismique, ne modifient que négligemment l'hodochrone de référence.

Dans le reste de cet article, nous appellerons une phase un pointé qui ne porte pas l'étiquette *rejet*. Nos contraintes proviennent en fait d'une comparaison entre des phases : c'est pour cela qu'elles ne sont applicables que pour les scénarios *station* à au moins deux phases. De même, les contraintes portant sur les pointés ne fonctionnent qu'à partir de deux pointés par station.

Contrainte 1 (exclusivité) Deux pointés ne peuvent avoir la même étiquette excepté dans le cas où l'étiquette est celle de la classe rejet. Il s'agit de la contrainte globale *alldifferent_except_0* (”*tous_différents_sauf_0*”) où 0 représente notre étiquette *rejet*.

Contrainte 2 (ordre des phases) Les ondes P arrivent avant les ondes S. Il s'agit cette fois d'une contrainte globale spécifique à notre problème. Si p représente le nombre de pointés perçus par la station considérée et si (t_i, l_i) représente le i^{e}pointé, alors la contrainte peut s'écrire :

$$\forall i, j \in \{1, \ldots, p\} \text{ tels que } t_i < t_j,$$
$$\nexists\, (l_i, l_j) \text{ tel que } l_i \in \{Sg, Sn\} \text{ et } l_j \in \{Pg, Pn\}. \quad (1)$$

Contrainte 3 (temps maximal) La différence entre les dates d'arrivées de deux pointés étiquetés ne

doit pas dépasser un certain seuil $\Delta_{max}(l_1, l_2)$ où l_1 et l_2 sont des étiquettes différentes de *rejet* :

$$\forall i, j \in \{1, ..., p\} \text{ tels que } t_i < t_j \text{ et } l_i, l_j \notin \{rejet\} \\ |t_i - t_j| < \Delta_{max}(l_i, l_j). \quad (2)$$

Les deux premières contraintes sont fortement restrictives et réduisent considérablement l'espace de recherche du premier modèle. Sans ces contraintes, nous avions p^5 scénarios *station* possibles, où p est le nombre de pointés. A présent, avec seulement les deux premières contraintes, le nombre n de scénarios est donné par la relation :

$$n = 1 + 4 \times p + 8 \times C_p^2 + 8 \times C_p^3 + 4 \times C_p^4 \quad (3)$$

puisqu'il n'y a que 8 scénarios à 2 ou 3 phases et 4 scénarios à 4 phases possibles, complétés ensuite par des pointés rejetés. Ainsi, avec 4 phases, nous passons de 625 à 117 scénarios *station* possibles.

Le second solveur doit comparer deux scénarios *station* afin de déterminer s'ils sont géophysiquement compatibles. Tout scénario *station* est géophysiquement compatible avec un scénario *station* à une seule phase ou avec un scénario *station* qui n'a que des pointés étiquetés en *rejet* (appelé « scénario vide »). Ce n'est qu'à partir du second solveur que nous exploitons des informations avancées telles que le temps t_0 de l'événement et la distance δ entre la station et l'événement à partir de deux phases. Soit (a_{l_i}, b_{l_i}) les paramètres de la droite de l'hodochrone correspondant au type de phase l_i. Nous considérons deux phases (t_1, l_1) et (t_2, l_2), telles que $t_1 < t_2$ qui ne sont pas nécessairement consécutives. Alors t_0 et δ peuvent s'exprimer comme suit :

$$t_0 = \frac{a_{l_1} \times (t_1 + b_{l_2}) - a_{l_2} \times (t_2 - b_{l_1})}{a_{l_1} - a_{l_2}} \quad (4)$$

$$\delta = \frac{t_1 - t_2 - b_{l_1} + b_{l_2}}{a_{l_1} - a_{l_2}} \quad (5)$$

Par la nature de l'hodochrone et une estimation rapide de l'erreur, au plus les phases choisies seront éloignées, au plus t_0 et δ seront précis. Les contraintes du second modèle sont soit binaires soit unaires et prennent des scénarios *station* en argument :

Contrainte 4 (dates de l'événement similaires) La différence entre les dates de l'événement estimées de façon indépendante pour chacune des stations doit être inférieure à un seuil Δ_{t_0}. Soit t_0^1 and t_0^2 l'estimation du t_0 à partir du premier et du second scénario *station* :

$$\left|t_0^1 - t_0^2\right| < \Delta_{t_0}. \quad (6)$$

Contrainte 5 (antériorité des t_0) Les dates de l'événement estimées de façon indépendante pour chacune des stations doivent être antérieures à la première phase de chacune des deux stations. Si (t_1, l_1) représente la première phase de la station, alors $t_0 < t_1$ doit être vérifié.

Contrainte 6 (croissance de l'hodochrone) Si la première phase de la première station est antérieure à la première phase de la seconde station et qu'elles ont la même étiquette, alors la seconde phase de la première station doit aussi être antérieure à la seconde phase de la seconde station si elles ont la même étiquette. Supposons que le premier scénario *station* S_1 soit constitué de n pointés et que le second scénario *station* S_2 soit constitué de m pointés, alors :

$$\forall i, j \in \{1, ..., n\} \text{ et } \forall i\prime, j\prime \in \{1, ..., m\} \\ \text{tels que } t_i < t_j, t_{i\prime} < t_{j\prime}, l_i, l_j, l_{i\prime}, l_{j\prime} \notin \{\text{rejet}\}$$

$$(l_i = l_{i\prime}) \wedge (l_j = l_{j\prime}) \wedge (t_i < t_{i\prime}) \Rightarrow (t_j < t_{j\prime}). \quad (7)$$

Contrainte 7 (inégalité triangulaire) Il s'agit d'une règle géométrique contraignant la distance estimée des stations par rapport à l'épicentre de l'événement. Soit deux stations éloignées de l'épicentre des distances δ_1 et δ_2 et éloignées entre elles de la distance δ_{12}. Ces trois distances sont liées par les relations suivantes :

$$|\delta_1 - \delta_2| / \tau \leq \delta_{12} \leq (\delta_1 + \delta_2) \times \tau \quad (8)$$

où τ est un facteur d'assouplissement de la contrainte géométrique.

Contrainte 8 (absence des phases P_n et S_n) En dessous d'une certaine distance $\delta_{P_nS_n}$ avec l'épicentre, une station ne peut percevoir de phase P_n ou S_n. $\delta_{P_nS_n}$ est appelée "distance critique". Si S représente le scénario *station* courant et si on suppose que nous avons un prédicat *Contient(S)* dont la valeur est vraie si S contient une phase Pn ou Sn, alors la contrainte peut s'écrire :

$$Contient(S) \Rightarrow (\delta > \delta_{PnSn}). \quad (9)$$

Contrainte 9 (respect de l'hodochrone) Les différents pointés étiquetés sont replacés sur l'hodochrone afin de vérifier s'ils sont cohérents entre eux. Les deux pointés les plus éloignés sont utilisés pour déduire la distance entre l'épicentre et la station. A partir de cette distance, l'hodochrone nous permet de prédire le délai entre les autres phases. La différence entre le délai prédit et le délai observé doit être inférieure à un seuil $\Delta_{hodochrone}$. Soit $t_0(i, j)$ et $\delta(i, j)$ la date de

l'événement et sa distance à la station estimés à partir de la i^{e}et de la j^{e}phase, alors :

$$\forall i,j,k \in \{1,..,n\}$$
$$\text{tels que } t_i \neq t_j \neq t_k \text{ et } l_i, l_j, l_k \notin \{rejet\}$$

$$t_k - t_0(i,j) - a_{l_k} \times \delta(i,j) + b_{l_k} \leq \Delta_{hodochrone} (10)$$

Afin de relâcher un peu les contraintes, compte tenu qu'elles ne sont déduites que d'un hodochrone de référence, nous avons introduit différents seuils. Ces seuils ont une signification géophysique et peuvent être paramétrés avec l'aide des experts. Dans le reste de cet article, nous dirons que deux scénarios *station* sont incompatibles si au moins une des contraintes ci-dessus est violée.

Les contraintes unaires du second modèle semblent adéquates au premier modèle. Cependant, par un soucis de temps de calcul, nous ne calculons t_0 et δ qu'une seule fois durant le processus, et les contraintes du second modèle utilisent plus souvent ces valeurs. Notons aussi que les contraintes 4 et 7 sont redondantes (sans être redondantes au sens de la propagation [3]) puisque t_0 et δ sont tirés de la même équation.

Dans le cas de petits événements sismiques, c'est-à-dire avec un nombre de pointés limités, les solutions sont générées en quelques secondes. Malheureusement, dans le cas de séismes moyens ou forts, l'objectif de la demi-minute que nous nous sommes fixés est largement dépassé. Dans les cas les plus complexes, il faut jusqu'à une dizaine de minutes. Cela vient du fait que certains scénarios ne sont filtrés par aucune contrainte du second solveur, comme par exemple les scénarios à une seule phase ou les scénarios vides, ce qui a pour effet d'augmenter considérablement le nombre de combinaisons à tester. Afin d'éviter ce problème, nous avons introduit une contrainte globale qui décuple la vitesse de parcours de l'espace de recherche.

4 Contrainte globale

Afin d'accélérer le parcours de l'espace de recherche, nous avons introduit une contrainte globale proche de la subsumption en logique. La subsumption est une relation entre deux concepts A et B. Si A possède certaines propriétés et que A est plus général que B, alors B possède ces mêmes propriétés. On dit alors que A subsume B. Nous nous proposons d'avoir ce genre de relation entre deux scénarios *station* afin d'éliminer de nombreuses valeurs des domaines des variables du second modèle.

Prenons par exemple deux stations S_1 et S_2 qui perçoivent trois phases chacune. Notons $\times$ les pointés rejetés. Si le scénario $sc_1 = \mathrm{P}_g \times \mathrm{S}_g$ de S_1 n'est pas compatible avec le scénario $sc_2 = \mathrm{P}_g \times \mathrm{S}_n$ de S_2, alors sc_1 est aussi incompatible avec $sc_2\prime = \mathrm{P}_g\ \mathrm{P}_n\ \mathrm{S}_n$ de S_2. Plus généralement, si sc_1 et sc_2 sont incompatibles, alors tout scénario de S_1 construit à partir de sc_1 sera incompatible avec tous les scénarios de S_2 construits à partir de sc_2. La construction d'un scénario à partir d'un autre se fait en remplaçant des pointés rejetés par des phases (tout en respectant les contraintes 1 à 3). En revanche la réciproque n'est pas vraie : si sc_1 est compatible avec sc_2, les scénarios dérivés ne sont pas forcément compatibles entre eux. A cause de cela, nous acceptons de procéder à un filtrage sous-optimal : nous allons considérer que la réciproque est vraie afin de diminuer la complexité du parcours de l'espace de recherche, en acceptant en contrepartie de filtrer moins de scénarios. Notre filtrage aurait été optimal sans la considération de cette hypothèse de travail.

Dans l'exemple précédent, les scénarios sc_1 et sc_2 sont appelés "scénarios générateurs" tandis que le scénario $sc_2\prime$ est appelé "scénario subsumé". Un scénario générateur et l'ensemble de ses subsumés sont appelés "une famille". Le tableau 1 montre deux exemples de familles de scénarios *station*.

générateurs	$\times$ $\times$	P_g $\times$ $\times$ S_n
subsumés	P_g $\times$	P_g P_n $\times$ S_n
	P_n $\times$	P_g P_n S_g S_n
	S_g $\times$	P_g S_g $\times$ S_n
	S_n $\times$	P_g $\times$ P_n S_n
	$\times$ P_g	P_g $\times$ S_g S_n
	$\times$ P_n	
	$\times$ S_g	
	$\times$ S_n	

TAB. 1 – Exemples de scénarios *station* et de leurs subsumés. Le symbole $\times$ représente la classe rejet.

Ainsi, nous considérons que le scénario *station* vide subsume les scénarios *station* à une phase et que les scénarios *station* à deux phases subsument les scénarios à trois ou quatre phases. En effet, les cas des scénarios *station* vides et ceux à deux phases doivent être séparés dans la mesure où les contraintes 5 à 9 découlent principalement de l'hodochrone que nous ne pouvons exploiter qu'à partir de deux phases.

La subsumption diminue ainsi la complexité du problème. En effet, si le pointeur automatique détecte 6 pointés pour la station S_1 , 365 scénarios *station* sont à envisager dont 120 à deux phases. De même, si le pointeur automatique détecte 8 pointés pour la station S_2 , 985 scénarios *station* sont à envisager dont 224 à deux phases. Plaçons nous dans le pire des cas, c'est-à-dire celui dans lequel le premier solveur n'a filtré aucun scénario. Pour construire l'ensemble des scénarios *réseau* possibles compte tenu des stations S_1 et S_2, il faudra tester chacun des scénarios de S_1 avec chacun des scénarios de S_2. Sans la subsumption, cela revient à effectuer $365 \times 985 = 359525$ comparaisons,

alors qu'avec la subsumption, ce nombre de comparaisons est réduit à $121 \times 225 = 27225$, soit plus de 13 fois moins. En effet, 121 et 225 sont le nombre de scénarios *station* générateurs de S_1 et S_2, c'est-à-dire le nombre de scénarios *station* à deux phases auxquels s'ajoute le scénario *station* vide. En considérant un nombre de stations plus grand, l'apport de la subsumption est pleinement révélé.

L'introduction de cette contrainte globale a un impact sur l'architecture du système (figure 6). D'un côté, puisque seuls les scénarios *station* vides ou à deux phases sont nécessaires pour générer les scénarios *réseau*, nous avons bridé le premier solveur afin qu'il ne fournisse que ce genre de solutions, à savoir des scénarios générateurs. La production de ces scénarios est ainsi plus rapide à tel point que la parallélisation n'est plus nécessaire. D'un autre côté, le second solveur peut rester inchangé : il combinera les scénarios *station* générateurs entre eux pour obtenir les scénarios *réseau*. Cependant, en ne considérant que les scénarios générateurs, le second solveur ne fournit qu'un sous-ensemble des solutions possibles que nous appelons « familles de scénarios *station* ». Nous complétons donc ces familles en calculant les subsumés de chacun des scénarios générateurs à l'aide d'un troisième solveur.

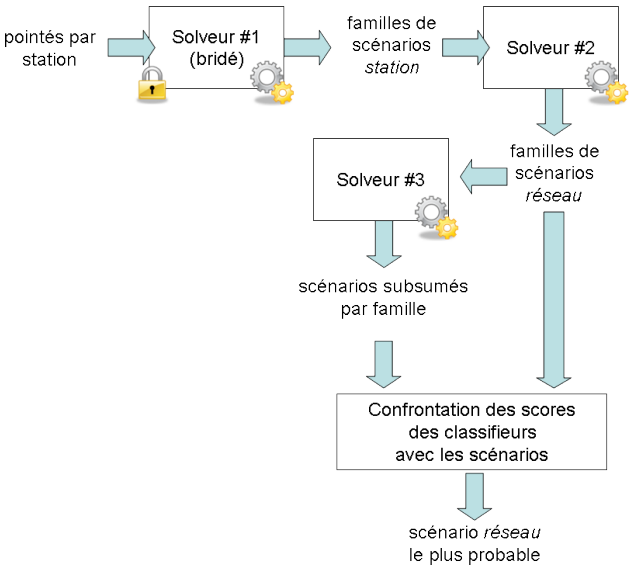


FIG. 6 – Nouvelle architecture du système.

Le troisième solveur doit uniquement construire les subsumés à partir des scénarios générateurs impliqués dans au moins une famille de scénarios *réseau*. Par souci d'efficacité de la contrainte globale, les contraintes utilisées par le troisième solveur vérifient juste la compatibilité des scénarios subsumés avec le scénario générateur. Le tableau 2 montre la nouvelle répartition des contraintes dans les trois solveurs.

Solveur	Rôle	Contraintes
#1	Calcul bridé des scénarios *station*	1, 2, 3
#2	Calcul des familles de scénarios *réseau*	4, 5, 6, 7, 8
#3	Calcul des scénarios subsumés *station*	1, 2, 3, 9

TAB. 2 – Répartition des contraintes

Station 1	Station 2	Station 3
Pg × Pn ×	Pg × Sn	× ×
Pg × Pn Sg ⋆	Pg Pn Sn ⋆	Pg ×
Pg × Pn Sn	Pg Sg Sn	Pn ×
Pn × × Sn	Pg × Sg ⋆	Sg ×
Pn Pg × Sn ⋆	Pg Pn Sg	Sn × ⋆
Pn × Pg Sn		× Pg
Pn × Sg Sn		× Pn
		× Sg
		× Sn

TAB. 3 – Familles de scénarios *station* pour trois stations. Les cellules grisées représentent les scénarios générateurs de chaque famille et les ceux marqués par ⋆ sont les plus probables de leur famille.

Cet arrangement en familles va rendre plus efficace le calcul des probabilités pour trouver le scénario *réseau* le plus probable : il s'agit d'une sorte de factorisation des scénarios *réseau*. En effet, pour chacune des familles de scénarios *station*, nous allons trouver le scénario le plus probable. Ensuite, pour chaque famille de scénarios *réseau*, nous allons multiplier entre elles les probabilités des ces scénarios *station* les plus probables. Ainsi, la redondance dans les calculs est éliminée. Prenons par exemple trois stations dont les familles de scénarios *station* sont répertoriées dans le tableau 3 : les scénarios les plus probables de chaque famille sont marqués du caractère ⋆ et les scénarios grisés sont les scénarios générateurs de chaque famille. Dans cet exemple, on suppose que tous les scénarios sont compatibles entre eux.

Sans l'agrégation en familles, il faudrait combiner les scénarios *station* entre eux et former ainsi 315 scénarios *réseau*, puis trouver le plus probable d'entre eux. A présent, avec le regroupement en familles, il suffit de trouver pour chaque famille de scénarios *réseau* le scénario le plus probable en prenant pour chaque famille de scéarios *station* qui la composent le scénario *station* le plus probable. Cela mène au final à la comparaison de seulement 4 scénarios *station* ce qui est plus rapide.

5 Expérimentations

Tous les paramètres de notre système (hodochrone, seuils) ont été fixés à l'aide des experts : en effet, ces paramètres ont un sens géophysiques et sont souvent interdépendants. Nous avons tenté de caractériser les performances de notre système, ce qui n'est pas facile compte tenu que les effets des contraintes du second modèle sont difficilement quantifiables.

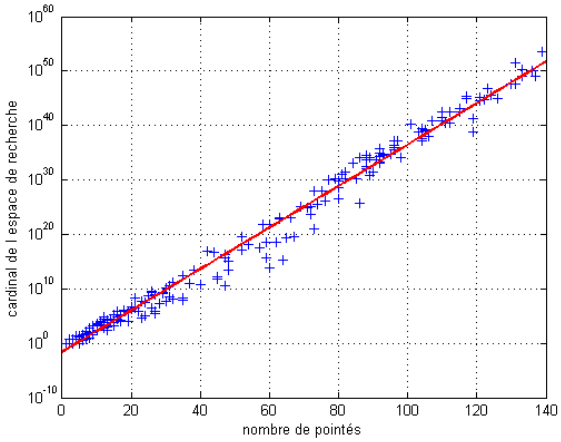


FIG. 7 – Estimation de la taille de l'espace de recherche en fonction du nombre de pointés (échelle log-log)

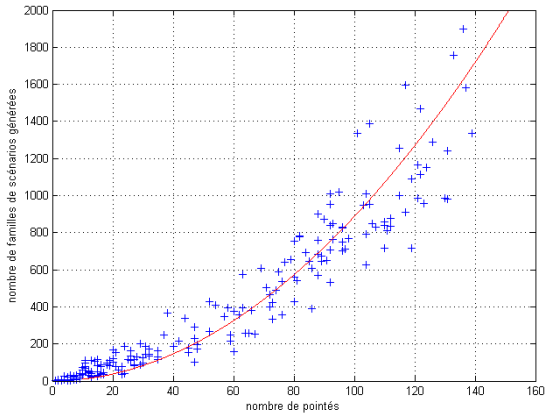


FIG. 8 – Estimation du nombre de scénarios *réseau* en fonction du nombre de pointés (échelle linéaire)

Le premier et le troisème modèle génèrent les solutions de façon presque instantanée. La génération des scénarios à 2 phases ne nécessite plus d'être vu comme un problème de satisfaction de contraintes tant la combinatoire est faible. En revanche, nous nous sommes intéressés au second modèle qui est confronté à un espace de recherche plus important. Il est possible d'estimer grossièrement la taille de son espace de recherche en multipliant les cardinaux des domaines des variables impliquées, c'est-à-dire en multipliant le nombre de scénarions *station* possibles pour chaque station entre eux. La figure 7 montre la taille de cet espace de recherche en fonction du nombre total de pointés. Après une simple régression, nous obtenons la relation suivante :

$$C \quad \approx 0.02 \times e^{0.88 \times p} \tag{11}$$

où C est la taille de l'espace de recherche et p le nombre de pointés. Cette relation montre que le cardinal de l'espace de recherche croît de façon exponentielle par rapport au nombre de pointé, même si la croissance est limitée par des coefficients assez faibles.

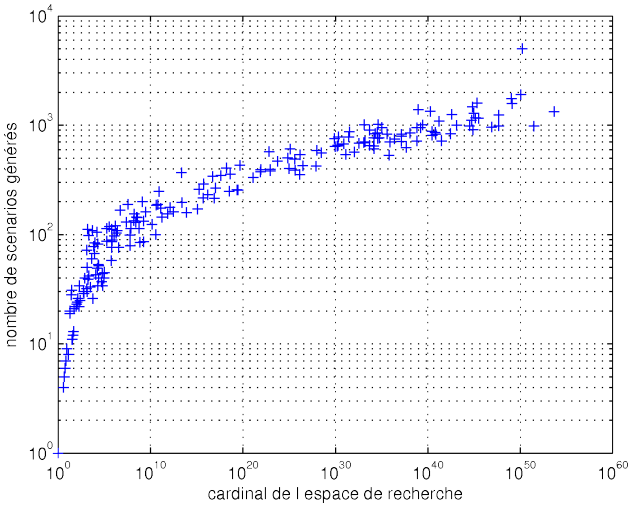


FIG. 9 – Nombre de familles de scénarios *réseau* générées en fonction du cardinal de l'espace de recherche. Echelle log-log.

Puisque le second modèle doit générer des familles de scénarios *station*, nous nous sommes intéressés au lien entre le cardinal de l'espace de recherche et le nombre familles générées. Il est important que ce nombre reste relativement faible afin de ne pas augmenter le coût des post-traitements comme la recherche du scénario le plus vraisemblable. La figure 9 montre que l'évolution suit une fonction croissante concave. Ainsi un espace de recherche plus grand ne va pas apporter nécessairement beaucoup plus de familles de scénarios *réseau* par rapport à un espace de recherche plus petit.

Enfin, nous avons tenté de caractériser le lien entre le temps de calcul total (comprenant les trois modèles) et le nombre de pointés. Cette relation est montrée dans la figure 10. Par régression, la relation s'écrit :

$$T \quad \approx 2.56 \times 10^{-7} \times p^{3.75} \tag{12}$$

où p est le nombre de pointés et T le temps de calcul en secondes. Puisqu'une alerte sismique doit être donnée sous trois minutes, nous pouvons déduire de cette dernière équation que le système pourra traiter le cas d'un séisme ayant généré 229 arrival times.

Ainsi, l'événement de 2005 qui a le plus grand espace de recherche a activé 42 stations et le pointeur auto-

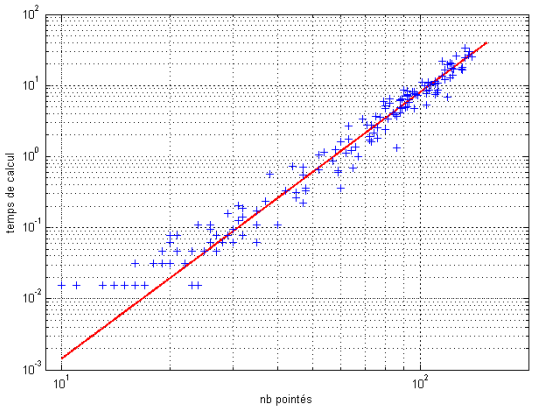


FIG. 10 – Temps de calcul (en secondes) en fonction du nombre de pointés (échelle log-log)

matique a généré 139 pointés. Pour cet événement, 1334 familles de scénarios *station* ont été générées en moins de 25 secondes. Ces familles, une fois dévéloppées, représentent plus de 1.6×10^{49} scénarios *réseau*. Ceci révèle pleinement l'avantage de la représentation en famille et la faiblesse du filtrage produit par les contraintes.

6 Conclusion

Nous avons présenté dans cet article un cas concret de l'utilisation des problèmes de satisfaction de contraintes dans le cas où l'ensemble des solutions est nécessaire. Nous avons présenté un système dont l'architecture originale s'intègre dans une chaîne de traitement existante. Le but est d'automatiser la procédure de localisation des événements sismiques et notre première tâche a été de déceler une des faiblesses du système proposé dans [5].

Notre amélioration consiste à générer tous les scénarios géophysiques possibles et d'exhiber le plus vraisemblable. La génération des scénarios se fait à l'aide d'un enchaînement de solveurs qui utilisent des contraintes fournies par les experts du DASE. Afin d'accélérer la production des scénarios, nous avons introduit une contrainte globale qui s'avère fortement efficace en diminuant l'espace de recherche durant le parcours de celui-ci. En contrepartie, pour mettre en place cette contrainte globale, nous avons accepté d'avoir un filtrage sous-optimal qui produira quelques scénarios en trop.

En pratique, le nombre de scénarios générés est parfaitement raisonable et notre système les calcule en quelques secondes. Dans le pire des cas étudiés, 25 secondes sont nécessaires pour produire 1334 familles de scénarios *réseau*.

Références

[1] James F. Allen. Maintaining knowledge about temporal intervals. *Commun. ACM*, 26(11) :832–843, 1983.

[2] J. Carrive, F. Pachet, and R. Ronfard. Clavis : a temporal reasoning system for classification of audiovisual sequences. In *Proceedings of Content-Based Multimedia Information Access (RIAO) Conference*, Paris, France, 2000.

[3] C. W. Choi, J. H.M. Lee, and P. J. Stuckey. Removing propagation redundant constraints in redundant modeling. *ACM Trans. Comput. Logic*, 8(4) :23, 2007.

[4] Vipin Kumar. Algorithms for constraint satisfaction problems : A survey. *AI Magazine*, 13 :32–44, 1992.

[5] A. Larue, D. Mercier, L. Cornez, F. Suard, J. Guilbert, and C. Maillard. Automatic classification of seismic phases for event location procedure. In *Proceedings of the 31st General Assembly of the European Seismological Commission*, 2008.

[6] A. Liret, N. Ramaux, N. Fontaine, M. Dojat, and F. Pachet. N-ary constraints for scenario recognition. Workshop on n-ary constraints, ECAI-98, 1998.

[7] Pascal Van Hentenryck. *Principles and Practice of Constraint Programming*. MIT Press, Cambridge, MA, USA, 1995.

Relaxation de contraintes globales pour la résolution de nurse-rostering problems

Jean-Philippe Métivier **Patrice Boizumault** **Samir Loudni**

GREYC (CNRS - UMR 6072) – Université de Caen
Boulevard du Maréchal Juin
14000 Caen
jmetivie@info.unicaen.fr {patrice.boizumault,samir.loudni}@unicaen.fr

Résumé

Les *nurse-rostering problems* consistent à affecter, pour chaque infirmière, une équipe sur une période de temps fixée en respectant des règles spécifiant les besoins de l'hôpital et la législation du travail. Ces problèmes sont généralement difficiles à résoudre car de grande taille et surcontraints. L'objectif de cet article est de montrer comment ces problèmes peuvent être modélisés et résolus à l'aide de contraintes globales relaxées dans des temps comparables aux méthodes ad'hoc utilisées.

1 Introduction

Les problèmes d'affectation d'infirmières (NRPs, *nurse-rostering problems*) consistent à affecter, pour chaque infirmière, une équipe (matin, soir, nuit, repos, ...) sur une période de temps fixée (horizon) en respectant des règles spécifiant les besoins de l'hôpital et la législation du travail. De tels problèmes sont généralement difficiles à résoudre car de grande taille et surcontraints. Le site ASAP[1] de l'Université de Nottingham [4] recense un ensemble de NRPs réels, ainsi que les méthodes utilisées jusque-là pour les résoudre. Ces problèmes sont souvent résolus par des méthodes de Recherche Opérationnelle (RO) ad'hoc, la plupart incluant des étapes de pré-taitement.

Dans le cadre d'une approche CSP, les contraintes globales procurent une modélisation élégante et un filtrage très performant pour la résolution des NRPs. Mais, ces problèmes sont par nature surcontraints : des coûts de violation sont associés à la non-satisfaction de certaines règles. L'objectif de cet article est de montrer comment ces problèmes peuvent être modélisés et résolus à l'aide de contraintes globales relaxées dans des temps comparables aux méthodes de RO ad'hoc utilisées.

De manière générale, on peut distinguer deux grandes sortes de règles : celles qui imposent des bornes sur le nombre d'infirmières, d'équipes,... qui seront traduites par des contraintes `Gcc` [15] (ou une de ses versions relaxées Σ-`Gcc` [10]) et celles qui imposent, pour chaque infirmière, des contraintes sur l'enchaînement des équipes qui seront traduites par des contraintes `Regular` [13] (ou une de ses versions relaxées `costRegular` [2]).

La Section 2 fait un panorama synthétique des NRPs et présente un exemple introductif que nous estimons être représentatif des différents NRPs. Dans la Section 3, nous présentons la/les relaxation(s) des contraintes globales `Gcc` et `Regular`. À l'aide de cet exemple, nous montrons (Section 4) comment les NRPs peuvent être modélisés de manière concise et élégante à l'aide des contraintes globales relaxées.

Les 2 sections suivantes sont consacrées à la résolution des NRPs ainsi modélisés. Tout d'abord, nous définissons (Section 5) la contrainte globale `RegularCount`, et sa version relaxée, qui combine les contraintes `Regular` et `Atleast`/`Atmost` afin d'obtenir un filtrage nettement plus performant que celui réalisé séparément par chacune de ces contraintes. Dans la Section 6, nous décrivons la méthode retenue pour nos expérimentations. Il s'agit d'une méthode de type VNS (Variable Neigbourhood Search) où la reconstruction des solutions s'effectue à l'aide d'un LDS (Limited Discrepancy Search) combiné avec le filtrage précédemment décrit.

Dans la Section 7, nous comparons expérimentale-

[1] Automated Scheduling, Optimisation and Planning, http ://www.cs.nott.ac.uk/˜tec/NRP/

ment notre approche avec les méthodes de RO utilisées sur un ensemble de problèmes réels extraits du site du ASAP. Nos expérimentations montrent que, malgré la flexibilité et la généricité de notre approche, des solutions de bonnes qualités peuvent être obtenues rapidement et que cette approche est compétitive par rapport aux méthodes ad'hoc développées jusque là.

2 Problème d'affection d'infirmières

2.1 Description du problème

Un NRP consiste à affecter, pour chaque infirmière, une équipe (*shift*) sur une période de temps fixée (*planning horizon*) en respectant des règles spécifiant les besoins de l'hôpital et la législation du travail. Le but est de trouver un affectation complète satisfaisant au mieux les contraintes du problème.

À chaque équipe est associée une période de temps durant laquelle une infirmière est en service. On distingue généralement les équipes suivantes : l'équipe du matin M, du soir E et celle de nuit N. Pour représenter les périodes de repos, on introduit l'équipe de repos R. On notera par *2-shifts problem*, les problèmes constitués des équipes M et N de durée égale à 12 heures, et par *3-shifts problem*, ceux constitués des équipes M, E et N de durée de 8 heures.

Chaque équipe doit respecter des exigences en terme de nombre d'infirmières requis et de qualification de celles-ci (contraintes d'équipes "*shift constraints*"). Par ailleurs, le planning d'une infirmière doit tenir compte de règles sur les séquences de travail (longueur maximale de ces enchaînements, type d'enchaînement,...) et des préférences des infirmières (contraintes d'infirmières "*nurse constraints*"). Notons qu'une infirmière ne doit pas travailler dans plus d'une équipe par jour. Parmi ces exigences, certaines doivent être absolement respectées (contraintes d'intégrité) alors que d'autres peuvent ne pas être satisfaites (contraintes de préférences dont les coûts de violation sont donnés dans la spécification même du problème).

2.2 Exemple introductif

Considérons le problème à *3-shifts* constitué de 8 infirmières travaillant sur un horizon de 21 jours et devant respecter les règles suivantes[2].

- Contraintes d'intégrité
  - (I1) Les équipes du matin M et du soir E sont composées de 2 infirmières et celle de nuit N d'une infirmière ;
  - (I2) Une infirmière ne doit pas travailler plus de quatre fois en équipe de nuit.
  - (I3) Sur une période de 21 jours, une infirmière doit au moins avoir sept repos.
  - (I4) Une infirmière doit avoir au moins un dimanche de libre toutes les trois semaines.
  - (I5) Une infirmière doit au plus travailler trois nuits consécutives.
  - (I6) Deux équipes successives doivent être séparées d'au moins 8 heures de repos.
- Contraintes de préférences
  - (P1) Pour une infirmière, un repos isolé est pénalisé par un coût de 100.
  - (P2) Une infirmière doit travailler de 4 à 8 fois en équipe du matin et de 4 à 8 fois dans celle du soir. Tout manque ou excès (δ) est pénalisé par un coût de ($10\times\delta$).
  - (P3) Une infirmière ne doit pas travailler plus de 4 jours consécutifs. Chaque jour supplémentaire entraine une pénalité de 1000.
  - (P4) Chaque nuit isolée entraîne une pénalité de 100.
  - (P5) Deux équipes successives doivent être séparées d'au moins 16 heures de repos. Tout écart inférieur est pénalisé d'un coût de 100.

La régle (I1) est une *shift constraint* alors que les autres sont des *nurse constraints*.

3 Relaxation de contraintes globales

3.1 Relaxer une contrainte globale

Relaxer une contrainte globale [17] c'est :

- **définir une sémantique de violation** permettant de quantifier le degré de violation de la contrainte globale ;
- **déterminer le niveau de cohérence** que l'on souhaite maintenir ;
- **proposer un test de cohérence et un algorithme de filtrage** associé.

Il peut exister plusieurs relaxations d'une même contrainte globale selon la sémantique de violation considérée. Le niveau de cohérence souhaité est bien entendu global, mais il peut être plus faible en l'absence d'algorithmes de test de cohérence et de filtrage performants. Par exemple, le test de cohérence est un problème NP-Complet pour la relaxation de la contrainte globale `AllDifferent` pour la sémantique de violation basée décomposition [14, 10], alors que ce même test est de complexité polynomiale faible pour la relaxation de `AllDifferent` pour la sémantique de violation basée variables [9].

[2] Ces règles, extraites de problèmes réels (cf. [4]), nous semblent représentatives des NRP.

3.2 Relaxer la contrainte globale Gcc

3.2.1 Global Cardinality Constraint

La contrainte globale `Gcc` [15] impose qu'un ensemble de variables prenne ses valeurs de façon à respecter des bornes inférieures et supérieures sur le nombre de fois que ces valeurs peuvent êtres prises.

Défintion 1. *Soit $\mathcal{X}$=$\{X_1,\ldots,X_n\}$, D_i le domaine de la variable X_i et $Doms$=$\cup_{X_i\in\mathcal{X}} D_i$. Soit v_j∈$Doms$ et l_j et u_j les bornes inférieures et supérieures de v_j. `Gcc`$(\mathcal{X},[l_1,\ldots,l_m],[u_1,\ldots,u_m])$ admet une solution ssi il existe une instanciation complète $\mathcal{A}$ telle que :*

$$\forall v_j \in Doms,\ l_j \leq \mid \{X_i \in \mathcal{X} \mid X_i = v_j\} \mid \leq u_j$$

La contrainte `Gcc` se modélise par un réseau pour lequel l'existence d'un flot maximal de valeur n permet de tester la cohérence ([15]). La recherche des composantes fortement connexes dans le graphe résiduel permet de filtrer toutes les valeurs non viables. Le test de cohérence se fait en $O(n\times m)$ et le filtrage en $O(n+m)$ (avec m égal au nombre d'arcs du réseau).

Enfin, toute contrainte `Gcc`(X,l,u) peut se décomposer en une conjonction de contraintes `Atleast` et `Atmost` portant sur les valeurs de $Doms$:
$\bigwedge_{v_j\in Doms}(\texttt{Atleast}(X,v_j,l_j) \wedge \texttt{Atmost}(X,v_j,u_j))$

3.2.2 La contrainte globale Σ-Gcc

Σ-`Gcc` est une version relaxée de `Gcc` selon la sémantique de violation basée décomposition notée μ_{dec} [10]. Σ-`Gcc` autorise le non respect des bornes inférieures et supérieures moyennant un coût de violation qui est fonction de l'écart aux bornes et du poids associé à celles-ci.

À chaque valeur v_j est associée un poids $\varphi_j^{atleast}$ (resp. φ_j^{atmost}) à la borne inférieure l_j (resp. supérieure u_j). On calcule le manque (resp. l'excès) grâce à la fonction $s(\mathcal{X},v_j)$ (resp. $e(\mathcal{X},v_j)$).

$$s(\mathcal{X},v_j) = max(0, l_j - \mid\{X_i \in \mathcal{X} \mid X_i = v_j\}\mid)$$
$$e(\mathcal{X},v_j) = max(0, \mid\{X_i \in \mathcal{X} \mid X_i = v_j\}\mid - u_j)$$

La violation de la contrainte Σ-`Gcc` pour la sémantique basée décomposition μ_{dec} est définie par :

$$\mu_{dec}(\mathcal{X}) = \sum_{v_j\in Doms} (s(\mathcal{X},v_j)\times\varphi_j^{atleast} + e(\mathcal{X},v_j)\times\varphi_j^{atmost})$$

Défintion 2 ([10])**.** *Soit z une variable objectif. Une contrainte Σ-`Gcc`$(\mathcal{X},l,u,\varphi^{atleast},\varphi^{atmost},z)$ admet une solution ssi il existe une instanciation complète $\mathcal{A}$ telle que $\mu_{dec}(\mathcal{A}) \leq max(D_z)$*

La modélisation d'une contrainte Σ-`Gcc` [10] s'effectue en ajoutant au réseau de `Gcc` des arcs de violation traduisant le manque ou l'excès pour chaque valeur. Ainsi sont ajoutés les arcs de violation suivants :

$$A_{shortage} = \{(s,v_j) \text{ avec } d=0, c=l_j, w=\varphi_j^{atleast} \mid v_j \in Doms\}$$
$$A_{excess} = \{(v_j,t) \text{ avec } d=0, c=\infty, w=\varphi_j^{atmost} \mid v_j \in Doms\}$$

L'existence d'un flot de valeur $max(n,\sum_{v_j\in Doms} l_j)$ de poids inférieur à $max(D_z)$ dans le nouveau réseau permet de caractériser la cohérence de Σ-`Gcc`. De la même manière, on peut caractériser la viabilité d'une valeur (X_i,v_j) par l'existence d'un flot passant par l'arc (X_i,v_j) de poids inférieur à $max(D_z)$. Le test de cohérence peut être réalisé en $O(max(n,\sum_{v_j\in Doms} l_j)\times n\cdot log(n))$ et le filtrage en $O(n\cdot d\times n\cdot log(n))$ [10].

Enfin, la contrainte `softGcc` [18] est un cas particulier de Σ-`Gcc` où tous les poids $\varphi_j^{atleast}$ et φ_j^{atmost} sont égaux à 1.

3.3 Relaxer la contrainte globale Regular

3.4 Regular

Soit M=$\{Q,\Sigma,\delta,q_0,F\}$ un automate fini déterministe avec Q l'ensemble des états, Σ un alphabet, δ l'ensemble des transitions définies sur $Q\times\Sigma\rightarrow Q$, q_0 l'état initial et F l'ensemble des états finaux de M. La contrainte `Regular`[13] associée à l'automate M impose qu'un mot constitué d'une séquence de n variables appartienne au langage reconnu par l'automate M.

Défintion 3. *Soit M un automate déterministe, $L(M)$ le langage régulier associé à M et $\mathcal{X}$=$X_1,\ldots,X_n$ une séquence de n variables. La contrainte `Regular`$(\mathcal{X},M)$ admet un solution ssi :*

$$\exists (v_1,\ldots,v_n) \in D_1\times\cdots\times D_n \text{ t.q. } (v_1,\ldots,v_n) \in L(M)$$

La contrainte `Regular` est modélisée par un graphe dirigé de couches dont les sommets de chaque couche correspondent aux états de l'automate M et les arcs représentent les couples (variable/valeur) [13].

Le graphe en couches $\mathcal{G}$ est définit de la manière suivante :

$$\mathcal{G} = \{V, A\}$$
$$V = V_0 \cup \cdots \cup V_n \cup \{t\}$$
$$A = A_0 \cup \ldots A_n \cup A_t$$
$$\forall i \in \{0,\ldots,n\}, \text{ on a :}$$
$$V_i = \{q_l^i \mid q_l \in Q\}$$
$$A_i = \{(q_l^i, q_m^{i+1}, v) \mid v \in D_i, \delta(q_l,v) = q_m\}$$
$$A_t = \{(q_l^n, t) \mid q_l \in F\}$$

Dans ce graphe, tout chemin du sommet représentant l'état initial q_0^0 dans la première couche vers un sommet modélisant un état final q_i^n (avec q_i∈F) dans

la dernière couche correspond à une solution pour `Regular`. De la même manière, on peut caractériser la viabilité d'une valeur (X_i,v_j) par l'existence d'un tel chemin passant par l'arc (X_i,v_j). Le test de cohérence et le filtrage sont mis en œuvre par un parcours en largeur du graphe en $O(m)$ (avec m égal au nombre d'arcs du graphe).

3.5 costRegular

`costRegular` [2] est une version relaxée de `Regular` qui permet de modéliser le fait que certaines transitions de l'automate engendrent un coût si elles sont utilisées. Du point de vue de la modélisation, les coûts de ces transitions sont directement reportés sur les arcs associés dans le graphe de couches. Pour la sémantique de violation retenue μ_{reg}, le coût d'une instanciation complète $\mathcal{A}$ est la somme des coûts des arcs appartenant au chemin associé à cette instanciation

Défintion 4. *Soit M un automate déterministe et soit z une variable objectif. Une contrainte* `costRegular`*$(\mathcal{X},M,z)$ admet une solution ssi il existe une instanciation complète $\mathcal{A}$ telle que $\mu_{reg}(\mathcal{A}) \leq max(D_z)$.*

Le test de cohérence, associé à la sémantique de violation μ_{reg}, se ramène à la recherche d'un chemin de poids minimal dans le graphe associé. De même, on peut caractériser la viabilité d'une valeur (X_i,v_j) par l'existence d'un chemin de poids inférieur à $max(D_z)$ passant par l'arc (X_i,v_j). Le nouveau graphe de couches restant acyclique, on peut à nouveau utiliser un parcours en largeur du graphe. Ainsi le test de cohérence et le filtrage peuvent être effectués en $O(m)$.

4 Modélisation du problème

Dans cette section, nous montrons comment le problème décrit en Section 2 peut être modélisé de façon concise et élégante à l'aide des contraintes globales relaxées. De manière générale, on peut distinguer 2 sortes de règles :

- celles qui imposent des bornes sur le nombre d'infimières, d'équipes, ..., qui seront traduites par des contraintes `Gcc` ou Σ-`Gcc` ;
- celles qui imposent, pour chaque infirmière, des contraintes sur l'enchaînement des équipes qui seront traduites par des contraintes `Regular` ou `costRegular`.

4.1 Variables et domaines

Soit l'ensemble des jours J={1,2,...,21} et l'ensemble des infirmières I={1,2,...,8}. À chaque infirmière $i \in I$ et chaque jour $j \in J$ est associée une variable X_j^i. Tous les domaines initiaux sont identiques $D_i^j = \{M, E, N, R\}$, et les valeurs de *Doms* seront considérées dans cet ordre pour les contraintes `Gcc`[3]. Il en sera de même pour les bornes et les coûts associés.

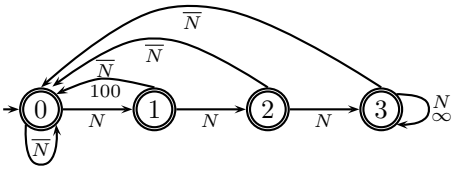


FIG. 1 – Automate A_1

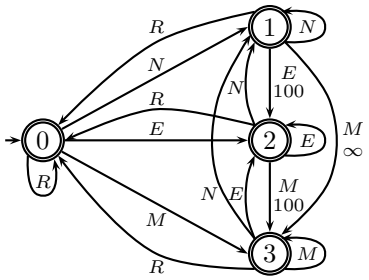


FIG. 2 – Automate A_2

4.2 Contraintes d'équipes

Les contraintes issues de la règle (I1) définissent les capacités de chaque équipe et seront modélisées grâce à la contrainte `Gcc`.

$$\boxed{\forall j \in J,\ \texttt{Gcc}([X_j^1,\ldots,X_j^8],[2,2,1,0],[2,2,1,8])}$$

4.3 Contraintes d'infirmières

i) Les règles suivantes peuvent être exprimées sous forme de `costRegular` en définissant les automates[4] suivants :

- l'automate A_1 associé aux règles (I5) et (P4) restreignant les enchaînements d'équipes de nuits (contrainte α) ;
- l'automate A_2 représentant les règles (I6) et (P5) contraignant l'écart minimal entre deux journées travaillées (contrainte β) ;
- l'automate A_3 décrivant les régles (P1) et (P3) gérant la longueur d'une séquence de travail (contrainte γ).

On obtient alors les contraintes suivantes :

$$\boxed{\begin{array}{ll} \forall i \in I,\ \texttt{costRegular}([X_1^i,\ldots,X_{21}^i],A_1,z_i^{A_1}), & (\alpha) \\ \forall i \in I,\ \texttt{costRegular}([X_1^i,\ldots,X_{21}^i],A_2,z_i^{A_2}), & (\beta) \\ \forall i \in I,\ \texttt{costRegular}([X_1^i,\ldots,X_{21}^i],A_3,z_i^{A_3}), & (\gamma) \end{array}}$$

[3] i.e. l=[l_M,l_E,l_N,l_R]

[4] La notation $\overline{N}$ signifie Σ\{N}

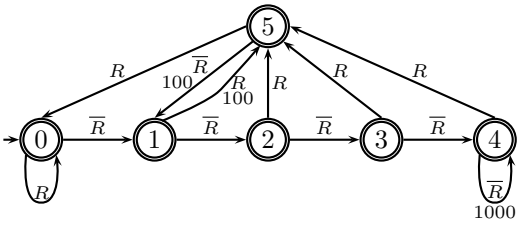


FIG. 3 – Automate A_3

ii) Les règles (I2) et (I3), qui sont des contraintes d'intégrité, peuvent être exprimées sous forme de `Gcc`. Toutefois, en associant des coûts infinis à ces contraintes, les règles (I2), (I3) et (P2) se modélisent grâce à un Σ-`Gcc` :

$\forall i \in I$, Σ-`Gcc`([X_1^i,...,X_{21}^i],[4,4,0,7], [8,8,4,21],[10,10,0,∞],[10,10,∞,0],z_i)

La règle (I2) ne définissant que la borne supérieure pour la valeur N (u_N=4), nous imposons que la borne inférieure associée à N soit nulle (l_N=0) ainsi que son coût de violation ($\varphi_N^{atleast}$=0). De même, pour la borne supérieure de R, issue de la règle (I3), nous imposons u_R=21 et φ_R^{atmost}=0.

iii) La règle (I4) peut être exprimée en utilisant une contrainte `Atleast` :

$\forall i \in I$, `Atleast`([X_7^i,X_{14}^i,X_{21}^i],1,R)

4.4 Fonction objectif

Soit $\mathcal{C}$ l'ensemble de toutes les contraintes globales précédemment décrites : à chaque contrainte relaxée c est associée une variable objectif z_c permettant de quantifier le coût de violation de celle-ci.

5 Intéraction entre contraintes globales

Malgré la puissance de filtrage de chaque contrainte globale, le manque de communication entre celles-ci réduit la qualité du filtrage final. En effet, chaque contrainte globale travaille à partir de sa représentation interne (graphe bipartite, réseau, ...) et n'exploite pas (ou que partiellement) les informations déduites par les autres contraintes globales. Dans les NRPs, un grand nombre de contraintes globales partagent des ensembles communs de variables. La plupart des contraintes d'infirmières portent sur l'intégralité du planning d'une infirmière.

Divers travaux ont déjà été menés pour exploiter l'interaction entre contraintes globales :

- `Cardinality Matrix Constraints` [16] combinant plusieurs `Gcc` sous forme d'une matrice ;
- `multi-costRegular` [8] fusionnant plusieurs `costRegular`.

Dans cette section, nous proposons une nouvelle contrainte globale `RegularCount` (et sa version pondérée) permettant de combiner une contrainte `Regular` avec plusieurs contraintes `Atleast`/`Atmost` portant sur une même valeur.

5.1 Exemple

Considérons les règles suivantes issues de l'exemple de la section 2 :

(I2) Une infirmière doit travailler au plus quatre nuits.

(I5) Une infirmière doit travailler au plus trois nuits consécutives.

(P4) Chaque nuit isolée est pénalisée d'un coût de 100.

Pour les besoins de l'exemple, nous considérons dans un premier temps la règle (P4) comme étant une contrainte d'intégrité :

(P*4) Une infirmière ne doit pas avoir de nuit isolée dans son planning.

Ainsi, la règle (I2) peut être modélisée grâce à une contrainte `Atmost` et les règles (I5) et (P*4) par un contrainte `Regular` :

$\forall i \in I$, `Atmost`([X_1^i,..., X_7^i],N,4)
$\forall i \in I$, `Regular`([X_1^i,..., X_7^i],$A_1^\star$)

L'automate $A_1^\star$ est obtenu en enlevant la transition de l'état 1 à l'état initial dans l'automate A_1.

Considérons à présent un planning de sept jours pour une infirmière i et les domaines réduits des variables associés : $D_{X_1^i}$=$D_{X_2^i}$=$D_{X_3^i}$=$\{N\}$, $D_{X_4^i}$=$\{R\}$ et $D_{X_5^i}$=$D_{X_6^i}$=$D_{X_7^i}$=$\{N,R\}$. En appliquant de manière successive les filtrages des contraintes `Atmost` et `Regular`, aucun retrait de valeur n'est effectué.

Mais la valeur (X_5^i,N) est non viable. Si X_5^i est instanciée à N alors X_6^i sera nécessairement instanciée à N (règle (P*4)), ce qui engendre une incohérence (la règle (I2) est insatisfaite). La valeur (X_5^i,N) aurait du être filtrée ce qui n'est pas le cas. Ce raisonnement peut être réitéré pour les valeurs (X_6^i,N) et (X_7^i,N).

Dans ce qui suit, nous montrons qu'en tenant compte des interactions entre les contraintes `Atmost` et `Regular`, il est possible de fournir un filtrage plus efficace.

5.2 La contrainte RegularCount

La contrainte `RegularCount` permet à une séquence de variables $\mathcal{X}$ de prendre des valeurs telles que le mot engendré appartient au langage reconnu par un automate, tout en respectant des bornes sur le nombre d'occurence d'une valeur v_j fixée.

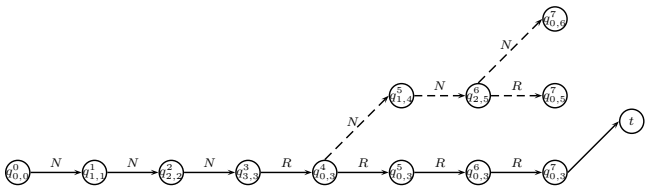


FIG. 4 – Réseau associé à l'exemple

Défintion 5. *Soit M un automate fini déterministe et $L(M)$ le langage reconnu par M. Soit $\mathcal{X}$=$X_1 \dots X_n$ une séquence de n variables, $v_j \in \Sigma$ et l_j (resp. u_j) une borne inférieure (resp. supérieure) sur le nombre de fois que la valeur v_j peut être prise. La contrainte* `RegularCount`*($\mathcal{X}$,M,v_j,[l_j,u_j]) admet une solution ssi :*

$$\exists (v_1, \dots, v_n) \in D_1 \times \dots \times D_n \text{ t.q. } (v_1, \dots, v_n) \in L(M) \\ \text{et } l_j \leq \mid \{X_i \in \mathcal{X} \mid X_i = v_j\} \mid \leq u_j$$

Ainsi, les règles (I2), (I5) et (P⋆4) peuvent être modélidées par la nouvelle contrainte `RegularCount` :

$\forall i \in I$, `RegularCount`($[X_1^i, \dots, X_7^i]$,$A_1^\star$,N,[0, 4])

5.2.1 Graphe associé à RegularCount

Soit v_j une valeur de Σ et l_j (resp. u_j) une borne inférieure (resp. supérieure) associée à la valeur v_j. L'idée principale est d'associer un compteur k à chaque sommet q_l^i de $\mathcal{G}$ pour comptabiliser le nombre de fois que la valeur v_j a été prise. Ainsi, à chaque sommet q_l^i sont associés k nouveaux sommets notés $q_{l,k}^i$, avec $k \in [0, \dots, n]$.

Les nouveaux sommets du graphe associé à la contrainte `RegularCount` sont définis par :

$$V = V_0 \cup \dots \cup V_n \cup \{t\}, \text{ avec :} \\ V_i = \{q_{l,k}^i \mid q_l \in Q, k \in [0..n]\}$$

Comme pour `Regular`, pour chaque transition entre deux couches du graphe est associé un arc reliant les sommets $q_{l,k}^i$ et $q_{m,k'}^{i+1}$ étiqueté par la valeur v_j. Toutefois, si à la valeur v_j est associé une paire de contraintes `Atmost`/`Atleast`, alors $k' = k+1$ sinon $k' = k$. Finalement, les sommets de la dernière couche ayant une valeur de compteur respectant les bornes imposées par les contraintes `Atmost`/`Atleast` (et correspondant à des états finaux) seront connectés au puit :

$$A_i = \{(q_{l,k}^i, q_{m,k'}^{i+1}, v) \mid v \in D_i, \delta(q_l, v) = q_m\} \\ \text{si } (v = v_j) \text{ alors } k' = k+1 \text{ sinon } k' = k \\ A_t = \{(q_{l,k}^n, t) \text{ si } q_l \in F \text{ et } l_j \leq k \leq u_j\}$$

La figure 4 représente le graphe associé à la contrainte `RegularCount`($[X_1^i, \dots, X_7^i]$,$A_1^\star$,N,[0, 4])) (exemple défini section 5.1).

5.2.2 Test de cohérence et filtrage

Le test de cohérence et le filtrage de `RegularCount` sont analogues à ceux de `Regular`. L'existence d'un chemin de $q_{0,0}^0$ à t permet de détecter la cohérence de la contrainte. L'existence d'un tel chemin utilisant l'arc (X_i,v) dans la couche correspondant à X_i permet de savoir si la valeur (X_i,v) est viable. Sur la figure 4, les arcs en pointillés indiquent les valeurs filtrées par la contrainte `RegularCount`. Ainsi, la valeur N est bien filtrée des domaines des variables X_5, X_6 et X_7 (exemple défini section 5.1).

5.3 La contrainte costRegularCount

Comme pour `CostRegular`, on souhaite modéliser le fait que les transitions impliquant une valeur particulière v_j engendrent un coût si elles sont utilisées. Les règles ci-dessous (issues de l'exemple de la section 2) permettent d'illuster une telle situation :

(I⋆2) Une infirmière doit travailler au plus quatre nuits, tout dépassement δ entraine une pénalité de $10 \times \delta$.

(I5) Une infirmière doit travailler au plus trois nuits consécutives.

(P4) Chaque nuit isolée est pénalisée d'un coût de 100.

Défintion 6. *Soit M un automate fini déterministe, $L(M)$ le langage associé, z une variable objectif. Soit $\mathcal{X}$ une séquence de n variables, $v_j \in \Sigma$, l_j (resp. u_j) la borne inférieure de poids $\varphi_j^{atleast}$ (resp. supérieure de poids φ_j^{atmost}) associé à v_j. La contrainte* `CostRegularCount`*($\mathcal{X}$,M,v_j,[l_j,u_j],[$\varphi_j^{atleast}$,φ_j^{atmost}],z) admet une solution ssi :*

$$\exists \mathcal{A} \in D_1 \times \dots \times D_n \text{ t.q } \mu_{reg}(\mathcal{A}) + s(\mathcal{A}, v_j) \times \varphi_j^{atleast} \\ + e(\mathcal{A}, v_j) \times \varphi_j^{atmost} \leq max(D_z)$$

Ainsi, les règles (I⋆2), (I5) et (P4) peuvent être modélisées par la contrainte `CostRegularCount` :

$\forall i \in I$:
`CostRegularCount`($[X_1^i, \dots, X_7^i]$,$A_1^\star$,N,[0, 4],[0, 10],$z_i^{A_1^\star}$))

Les sommets et les arcs du graphe correspondant à `CostRegularCount` sont construits de la même manière que pour `RegularCount`. Les arcs connectant les sommets de la dernière couche (correspondant à des états finaux) au sommet puits t sont remplacés par des arcs de violations permettant de modéliser le manque ou l'excès δ pour une valeur. Le coût de violation associé est $\delta \times \varphi^{atleast}$ (resp. $\delta \times \varphi^{atmost}$) pour la contrainte `Atleast` (resp. `Atmost`). Enfin, comme pour `CostRegular`, les coûts des transitions dans l'automate sont directement reportés sur les arcs associés dans

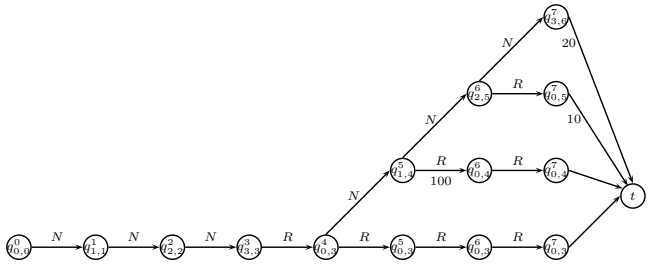


FIG. 5 – Réseau associé à l'exemple

le graphe de couches. La figure 5 donne le graphe correspondant à la contrainte `CostRegularCount`($[X_1^i,\ldots, X_7^i]$,$A_1^\star$,N,[0, 4],[0, 10],$z_i^{A_1^\star}$)).

Tester la cohérence revient à rechercher un chemin de poids inférieur à la valeur maximale de la variable objectif. Il en est de même pour le filtrage.

Complexité dans le pire cas : $O(n^2/2\times |\Sigma| \times |Q|)$

6 Variable Neighbourhood Search

La majorité des méthodes utilisées pour résoudre les NRPs sont soit des méthodes de RO ad'hoc, la plupart incluant des étapes de pré-taitement ou des méthodes de recherche locale combinant des techniques issues de la RO pour trouver une solution initiale (en résolvant à l'optimum un problème simplifié du problème de départ) pour la recherche locale.

Par nature, ces problèmes permettent de définir des structures de voisinages étendus (2-opt, swap de large portions des plannings d'infirmières,etc) en exploitant des informations spécifiques liées au problème.

La recherche à voisinage variable (VNS), [12], est une recherche à grands voisinages autorisant la taille de ceux-ci à varier afin de s'échapper des minima locaux. Pour cela, VNS utilise une *structure* de voisinage, c'est à dire un ensemble d'éléments $\{\mathcal{N}_1,\ldots,\mathcal{N}_{kmax}\}$ ordonnés par ordre croissant de la taille des voisinages engendrées.

Plusieurs versions de VNS ont été proposées. Variable Neighborhood Decomposition Search (VNDS), [3] permet d'effectuer une recherche uniquement sur un sous-problème déterminé en désaffectant un ensemble de k variables (k étant appelé la dimension du voisinage) et la reconstruction s'effectue alors uniquement sur le sous-espace de recherche contentant ces k variables. L'étape de reconstruction peut être effectuée par une recherche arborescente partielle.

6.1 VNS/LDS+CP

La méthode VNS/LDS+CP, [7], est une extension de l'algorithme VNDS effectuant la reconstruction des variables grâce à une méthode arborescente partielle (LDS) combinée avec des mécanismes de filtrage (CP). Toutefois, l'exploration de (très) grands voisinages pouvant rapidement devenir prohibitive en temps, nous avons retenu la recherche partielle LDS.

L'algorithme 1 décrit le pseudo-code de VNS/LDS+CP, avec $\mathcal{C}$ l'ensemble des contraintes, k_{init} (resp. k_{max}) le nombre minimal (resp. maximal) de variables à désaffecter et δ_{max} la valeur maximale de discrepancy lors de la phase de reconstruction avec LDS.

Algorithm 1: Algorithme VNS/LDS+CP.

```
1  function VNS/LDS+CP(X, C, k_init, k_max, δ_max)
2  begin
4      s ← genPremiereSolAvecLDS(X)
6      k ← k_init
8      while (k < k_max) ∧ (not timeout) do
10         X_unaffected ← H_vois(N_k, s)
12         A ← s\{(x_i = a) s.t. x_i ∈ X_unaffected}
14         s' ← NaryLDS(A, X_unaffected, δ_max, V(s), s)
16         if V(s') < V(s) then
18             s ← s'
20             k ← k_init
22         else k ← k + 1
24     return s
25 end
```

L'algorithme part d'une solution initiale s générée par la méthode LDS. Un sous-ensemble de k variables (avec k la dimension du voisinage) est sélectionné dans le voisinage N_k (i.e. l'ensemble des combinaisons de k variables parmi $\mathcal{X}$)(ligne 10). Une affectation partielle $\mathcal{A}$ est alors générée à partir de la solution courante s, en désaffectant les k variables sélectionnées ; les autres variables (i.e. non sélectionnées) gardent leur affectation dans s (ligne 12). $\mathcal{A}$ est alors reconstruite en combinant LDS et le filtrage. Si LDS trouve une solution voisine s' de meilleure qualité que la solution courante s (ligne 16), alors celle-ci devient la solution courante et k est réinitialisée à k_{init} (lignes 18-20). Sinon, k est incrémentée de 1 afin de s'échapper de ce minimum local(ligne 22). L'algorithme s'arrête dès que l'on a atteint la dimension maximale du voisinage à considérer k_{max} ou le timeout (ligne 8).

6.2 Gestion du voisinage

L'heuristique de choix de voisinage joue un rôle primordial dans la recherche, puisqu'elle détermine les sous-espaces à explorer afin de trouver des solutions de meilleure qualité. Notre heuristique de choix voisinage consiste à désaffecter complètement le planning d'une infirmière. Deux stratégies ont été mises en œuvre :

– l'heuristique `alea` basée sur un *tirage aléatoire* sur l'ensemble de tous les infirmières,

– l'heuristique `maxV` qui choisit l'infirmière dont le planning complet engendre le coût de violation le plus élevé.

6.3 LDS+CP

Notre heuristique de choix de variable sélectionne les variables par ordre croissant du ratio entre la taille du domaine et le degré. Les valeurs sont ordonnées selon l'ordre croissant des coûts engendrés par leurs affectations.

Le filtrage utilisé est celui des contraintes globales.

7 Expérimentations

Le solveur utilisé pour mener ces expérimentations a été développé en C++ et les tests ont été effectués sur PC P4-2.8Ghz. Les méthodes avec lesquelles nous nous comparons ayant été testées sur différents types de machines, nous exprimerons les temps de calcul obtenus en tenant compte du rapport de puissance entre la machine utilisée et la nôtre. Ainsi, pour une machine κ fois moins puissante que la nôtre, nous reporterons le temps obtenu divisé par κ

Pour VNS, k_{init} est le nombre de jours compris dans le planning d'une infirmière, et pour k_{max} 66% du nombre total de variables d'une instance. La discrepency est initialement fixée à 3 et la méthode est relancée avec une discrepancy augmentée après avoir exploré le voisinage de plus grande taille.

Les instances étudiées sont les suivantes :
– l'instance MILLAR [11] : 8 infirmères, 14 jours et 2 équipes possibles ;
– l'instance LLR [6] : 27 infirmières, 7 jours et 3 équipes possibles ;
– l'instance GPOST [4] : 8 infirmières, 28 jours et 2 équipes possibles ;
– l'instance IKEGAMI-3SHIFTS-DATA1 [5] : 25 infirmières, 30 jours et 3 équipes possibles.

N.B. : l'axe des ordonnées représente la valuation et celui des abscisses le temps en secondes.

7.1 Comparaison du filtrage de RegularCount

Nous avons modélisé chacune des instances LLR et GPOST de 2 façons différentes : à l'aide de `Atleast/Atmost+Regular` et à l'aide de `RegularCount`.

Comme le montrent les résultats présentés dans les figures 6 et 7, la contrainte `RegularCount` permet d'améliorer les performances de la recherche. Cette amélioration des performances est plus importante sur l'instance GPOST : cela est sûrement du au fait que les contraintes `RegularCount` remplacent dans GPOST un ensemble de contraintes `Regular/Atmost` portant

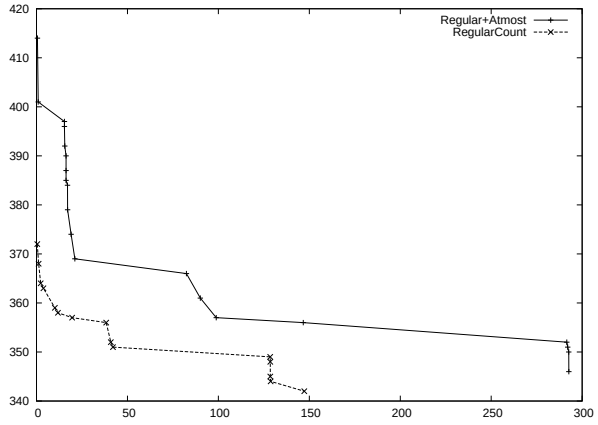


FIG. 6 – Comparaison de `Regular + Atmost/Atleast` et `RegularCount` sur l'instance LLR

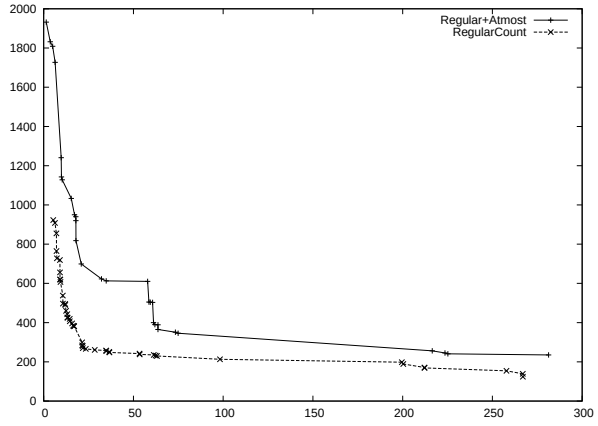


FIG. 7 – Comparaison de `Regular + Atmost` et `RegularCount` sur l'instance GPOST

sur 28 jours (et donc 28 variables) alors qu'elles ne portent que sur 7 jours dans l'instance LLR.

7.2 Comparaison avec une approche CSP

Dans [6] les auteurs proposent une approche basée CSP, pour l'instance LLR, qui utilise des contraintes binaires et des propagateurs spécifiques pour les contraintes n-aires.

La recherche s'effectue en deux étapes. Premièrement, une solution initiale est calculée en ne tenant compte que des contraintes d'intégrité. Dans une seconde temps, toutes les contraintes sont prises en compte et une procédure de recherche tabou améliore la solution initiale.

Grâce à cette méthode une solution de qualité 366 est trouvée en 16 secondes.

Nous trouvons une borne à 314 en environ 1 minute. Il est à noter que la première solution obtenue a pour qualité 372 et qu'une instanciation complète de coût

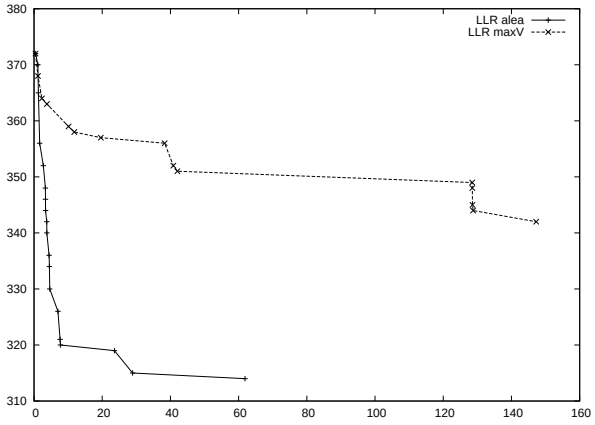


FIG. 8 – Instance LLR

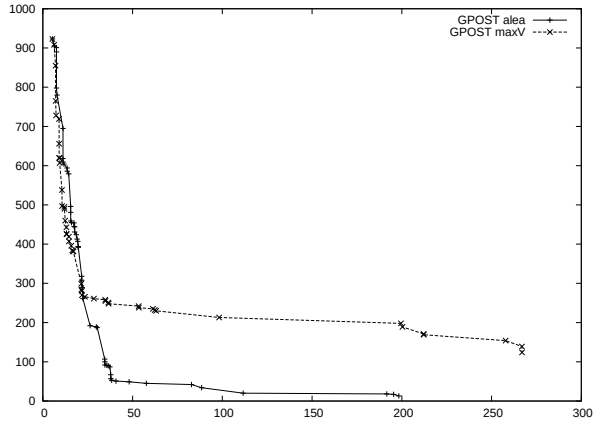


FIG. 9 – Instance GPOST

inférieur à 366 est obtenue en moins de 2 secondes avec les deux heuristiques de choix de voisinage : `alea` et `maxV`.

Ces résultats montrent que les contraintes globales relaxées permettent de trouver de bonnes premières solutions et d'améliorer rapidement la qualité de celles-ci grâce à la puissance de leur filtrage.

7.3 Comparaison avec des méthodes ad'hoc

7.3.1 L'instance MILLAR

Les premiers résultats obtenus pour MILLAR sont présentés dans [11]. La technique utilisée consiste à générer l'ensemble des motifs valides de longueur variant de 1 à 5 jours (une infirmières ne devant pas travailler plus de 5 jours consécutifs). À partir de ces motifs, un graphe dirigé est créé avec pour sommets les motifs et pour arcs les connexions entre les différents motifs. Ces arcs sont pondérés par le degré de violation provoqué par l'enchaînement de ces motifs.

Grâce à cette approche, une solution de coût 1690 est trouvée en environ 500 secondes sur IBM RISC6000/340[5].

Grâce à la résolution de sous-problème et d'une recherche tabou, une solution optimale (de coût 0) est trouvée en moins d'une seconde ([5]).

Notre solveur trouve l'optimum en 3 secondes.

NB : La borne supérieure utilisée dans [5] n'est pas indiquée, c'est pourquoi nous avons arbitrairement fixée notre borne initiale à 10000. Si celle-ci était fixée à une valeur plus faible, notre temps de calcul serait très fortement amélioré.

7.3.2 L'instance GPOST

Cette instance est résolue en 2 étapes (voir [4]). Dans un premier temps l'ensemble des plannings possibles pour chaque infirmière est calculé. Parmi ces plannings, seuls sont conservés ceux qui ont un degré de violation assez faible. Le planning général est alors calculé grâce à CPLEX. Le temps "d'assemblage" du planning général optimal est de 13 secondes (coût 5).

Pour cette instance nous avons borné le temps d'éxecution de notre solveur à 300 secondes. Nous obtenons les résultats présentés dans la figure 9.

Notre approche obtient en moins de 200 secondes une solution de coût 13.

Il est assez difficile de comparer nos temps de calcul avec ceux proposés dans [4] sans connaître le temps utilisé pour l'étape de prétraitement.

7.3.3 L'instance IKEGAMI-3SHIFTS-DATA1

Dans [5], avec une approche similaire à celle de MILLAR, les auteurs obtiennent une solution de coût 6 en 1851 minutes.

Pour des raisons pratiques, nous avons fixé le temps maximal de recherche à 1 heure.

Malgré la grande taille de cette instance (750 variables), les résultats présentés sont très encourageants (figure 10). En effet, l'heuristique de voisinage `maxV` nous fournit en 11 minutes et 11 secondes une solution de coût 63 . Cette solution est de bonne qualité car seules des contraintes de poids 1 sont insatisfaites.

8 Conclusions

Dans cet article, nous avons montré comment certains NRPs peuvent être modélisés à l'aide de contraintes globales relaxées et résolus dans des temps

[5]Le temps ici n'est pas normalisé

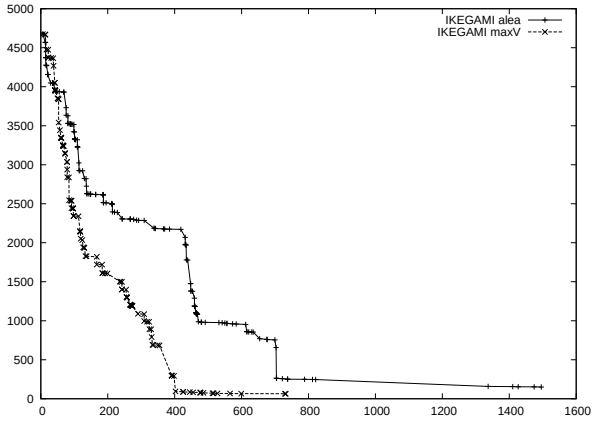


FIG. 10 – Instance IKEGAMI

comparables aux méthodes ad'hoc utilisées. Ces résultats sont à conforter par des expérimentations sur des instances de très grande taille (les instances : CHILD-A2 (41 infirmières × 42 jours × 5 équipes), ERRVH-A (51 infirmières × 42 jours × 8 équipes), ...). Les performances de notre méthode de recherche peuvent être améliorées i) en définissant des heuristiques de sélection de voisinage plus adaptées aux NRP (par exemple, ne pas désinstancier toutes les variables associées à une même infirmière), ii) en étendant la notion de soft arc-conhérence introduite par [1] pour les contraintes binaires afin d'améliorer la communication entre les contraintes globales.

Références

[1] Martin C. Cooper, Simon de Givry, Martí Sánchez, Thomas Schiex, and Matthias Zytnicki. Virtual arc consistency for weighted csp. In Dieter Fox and Carla P. Gomes, editors, *AAAI*, pages 253–258. AAAI Press, 2008.

[2] Sophie Demassey, Gilles Pesant, and Louis-Martin Rousseau. A cost-regular based hybrid column generation approach. *Constraints*, 11(4) :315–333, 2006.

[3] P. Hansen, N. Mladenovic, and D. Perez-Brito. Variable neighborhood decomposition search. *Journal of Heuristics*, 7(4) :335–350, 2001.

[4] http ://www.cs.nott.ac.uk/ tec/NRP/.

[5] Atsuko Ikegami and Akira Niwa. A subproblem-centric model and approach to the nurse scheduling problem. *Mathematical Programming*, 97(3) :517–541, 2003.

[6] Haibing Li, Andrew Lim, and Brian Rodrigues. A hybrid ai approach for nurse rostering problem. In *SAC*, pages 730–735. ACM, 2003.

[7] Samir Loudni and Patrice Boizumault. Combining vns with constraint programming for solving anytime optimization problems. *European Journal of Operational Research*, 191(3) :705–735, 2008.

[8] Julien Menana, Sophie Demassey, and Narendra Jussien. Relaxation lagrangienne pour le filtrage d'une contrainte-automate à coûts multiples. In Ammar Oulamara, editor, *ROADEF*, pages 43–44, 2009.

[9] Jean-Philippe Métivier, Patrice Boizumault, and Samir Loudni. Σ-alldifferent : Softening alldifferent in weighted csps. In *ICTAI (1)*, pages 223–230. IEEE Computer Society, 2007.

[10] Jean-Philippe Métivier, Patrice Boizumault, and Samir Loudni. Softening gcc and regular with preferences. In *SAC*, volume 3, pages 1392–1396, 2009.

[11] Harvey H Millar and Mona Kiragu. Cyclic and non-cyclic sheduling of 12h shift nurses by network programming. *European Journal of Operational Research*, 104 :582–592, 1996.

[12] N. Mladenovic and P. Hansen. Variable neighborhood search. *Computers And Operations Research*, 24 :1097–1100, 1997.

[13] Gilles Pesant. A regular language membership constraint for finite sequences of variables. In Mark Wallace, editor, *CP*, volume 3258 of *Lecture Notes in Computer Science*, pages 482–495. Springer, 2004.

[14] T. Petit, J-C. Régin, and C. Bessière. Specific filtering algorithms for over-constrained problems. In *CP'01*, volume 2239 of *LNCS*, pages 451–463, 2001.

[15] Jean-Charles Régin. Generalized arc consistency for global cardinality constraint. In *AAAI/IAAI, Vol. 1*, pages 209–215, 1996.

[16] Jean-Charles Régin and Carla P. Gomes. The cardinality matrix constraint. In Mark Wallace, editor, *CP*, volume 3258 of *Lecture Notes in Computer Science*, pages 572–587. Springer, 2004.

[17] Thomas Schiex. Soft constraint processing. *First International Summer School on Constraint Programming*, 2005.

[18] Willem Jan van Hoeve, Gilles Pesant, and Louis-Martin Rousseau. On global warming : Flow-based soft global constraints. *J. Heuristics*, 12(4-5) :347–373, 2006.

Problème d'équilibre des charges de travail dans l'affectation de patients aux infirmières. *

Pierre Schaus, Pascal Van Hentenryck, Jean-Charles Régin

Dynadec, One Richmond Square, Providence, RI 02906, USA
Brown University, Box 1910, Providence, RI 02912, USA
Université de Nice-Sophia Antipolis, France
pschaus@dynadec.com pvh@cs.brown.edu regin@polytech.unice.fr

Résumé

Cet article traite de l'affectation journalière d'enfants à des infirmières dans un hôpital. L'objectif est d'équilibrer la charge de travail des infirmières tout en satisfaisant diverses contraintes. Des travaux précédents ont proposé un modèle MIP pour ce problème, qui malheureusement rencontre des difficultés pour résoudre de grandes instances. De plus ce modèle MIP ne fait qu'approximer la fonction objectif. En effet la minimisation de la variance n'est pas une expression linéaire. Des modèles de programmation par contraintes (PC) sont présentés de complexités croissante permettant finalement de résoudre des instances de grande tailles avec des centaines de patients et d'infirmières en quelques secondes avec le système d'optimisation Comet. Les modèles de PC utilisent la contrainte globale spread pour minimiser la variance ainsi qu'une technique de décomposition du problème.

1 Introduction

Cet article traite le problème d'affectation journalière de bébés aux infirmières dans un hôpital décrit dans [5]. Dans ce problème, certains enfants ne demandent pas trop d'attention, par contre d'autres demandent une attention considérable. La quantité de travail requise par un enfant est appelée son acuité. Une infirmière sera chargée de s'occuper d'un groupe d'enfants et la somme des acuités des enfants dont elle a la charge est sa charge de travail. Il est essentiel que les charges de travail soient correctement équilibrées entre les infirmières. Cela permet d'assurer à la fois des soins de santé de meilleure qualité mais également de concevoir des horaires équitables entre les infirmières. La solution finale devra également satisfaire diverses contraintes additionnelles imposées par la législation en vigueur :

- Une infirmière ne peut travailler que dans une zone de l'hôpital alors que les enfants sont localisés dans p zones différentes.
- Une infirmière ne peut pas prendre en charge plus de $children^{\max}$ enfants.
- L'acuité totale d'une infirmière ne peut pas depasser $acuity^{\max}$.

L'objectif d'équilibre et les contraintes additionnelles rendent le problème difficile à résoudre. Etant donné que les infirmières ne peuvent travailler que dans une zone, le nombre d'infirmières attribuées à chaque zone a déjà une grande influence sur la qualité d'équilibre des charges de travail dans la solution finale.

Ce problème a initialement été résolu dans [5] avec un modèle MIP. Malheureusement les résultats en terme de qualité d'équilibre et temps de calcul ne sont pas satisfaisants. Dans cet article, nous présentons une série de modèles de programmation par contraintes de complexités croissantes. Notre dernier modèle permet de d'obtenir rapidement des solutions de grande qualité tout en passant très bien à l'échelle lorsque la taille de l'instance augmente.

L'organisation de l'article est la suivante. La Section 2 présente les instances proposées dans [5] et la Section 3 décrit le modèle MIP et ses limitations. La Section 4 rappelle la contrainte *spread* utilisée pour équilibrer les charges de travail ainsi que la caractérisation de son filtrage tel qu'implémenté dans Comet. La Section 5 présente le premier modèle de programmation par contrainte (PC). Ce modèle permet la résolution d'instances avec deux zones. La Section 6 présente une approche en deux temps où les infirmières sont d'abord

*traduction d'un article accepté à CPAIOR09

affectées aux zones avant de leur assigner des enfants. Finalement, la Section 7 montre que la deuxième étape peut être décomposée par zone sans perdre de garantie sur l'optimalité. Ce dernier modèle permet la résolution d'instances de grandes tailles avec des dizaines de zones et des centaines de patients.

2 Instances du problème

Un modèle statistique introduit dans [5] permettant de générer des instances similaires aux instances réelles rencontrées par les auteurs. Ce modèle statistique a également été utilisé dans leur article pour tester la robustesse de leur approche par rapport aux nombre d'infirmières, d'enfants, et de zones. Ce modèle ne comprend qu'un seul paramètre : le nombre de zones. L'acuité maximum par infirmière est fixée à $acuity^{\max} = 105$ et le nombre maximum d'enfants par infirmière est fixé à $children^{\max} = 3$. Le générateur d'instances fixe le nombre d'infirmières, le nombre d'enfants et leur acuité ainsi que la zone à laquelle il appartient. Voici précisément les différentes étapes suivies par le modèle statistique pour générer une instance :

- Le nombre de patients par zone suit une distribution de Poisson avec une moyenne 3.8 qui est décalée de +10.
- L'acuité Y d'un patient est obtenue en générant d'abord un nombre $X \sim Binomial(n = 8, p = 0.23)$ et ensuite en générant l'acuité $Y \sim Unif(10 \cdot (X+1), 10 \cdot (X+1) + 9)$.
- Le nombre total d'infirmière est obtenu en solutionnant une procédure *First Fit Decreasing* (FFD) dans chaque zone. Plus précisément, le nombre total est le nombre d'infirmières trouvées dans chaque zone par la procédure FFD. La procédure FFD commence par trier les patients par ordre décroissant d'acuité. Ensuite le patient avec la plus grande acuité est assigné à la première infirmière. Les patients suivants sont assignés successivement à la première infirmière pouvant l'accepter sans violer la contrainte d'acuité maximale et la contrainte du nombre maximum de patients par infirmière.

3 Le modèle MIP

Par manque de place, nous ne reproduisons pas ici le modèle MIP introduit dans [5] dans son entièreté. Nous donnons les principales variables du modèle ce qui est suffisent pour le cerner. Nous expliquons ensuite les limites de ce modèle et nous expliquons pourquoi une approche de PC peut pallier à ces limites. Le modèle MIP contient quatre familles de variables :

1. $X_{ij} = 1$ si l'enfant i est affecté à l'infirmière j et 0 sinon ;
2. $Z_{jk} = 1$ si l'infirmière j est assigné à la zone k et 0 sinon ;
3. $Y_{k,\max}$ est l'acuité maximum parmi toutes les infirmières de la zone k ;
4. $Y_{k,\min}$ est l'acuité minimum parmi toutes infirmières de la zone k.

Toutes ces variables sont liées par des contraintes linéaires pour satisfaire les contraintes du problème. La fonction objectif implémente ce que nous appelons le critère *range-sum* qui consiste en la minimisation de la somme des écarts entre l'acuité maximum et minimum dans les p zones, i.e.,

$$\sum_{k=1}^{p} (Y_{k,\max} - Y_{k,\min}).$$

Le modèle MIP comporte plusieurs limitations : La fonction objectif peut produire des solutions très inégales en termes de charge de travail. En effet elle tend à égaliser les charges de travail intra zone mais peut produire de grandes différences entre les temps de travail inter zone. Ce fait est illustré sur la Figure 1. Les charges de travail sont montrées en haut à droite de chacune des visualisations Comet. La solution de gauche est obtenue en minimisant le critère range-sum alors que la solution de droite est obtenue en minimisant la variance (nommée norme L_2 dans la section suivante). L'objectif range-sum est optimal dans la solution de gauche puisque les temps de travail à l'intérieur de chaque zone sont identiques. Malheureusement les infirmières de la première zone travaillent deux fois plus que les infirmières de la seconde zone. La solution de droite obtenue par minimisation de la variance est significativement meilleure. *Cette exemple illustre clairement que l'objectif que "toutes les infirmières devraient recevoir une même charge de travail" [5] n'est pas garanti avec le critère range-sum.*

Il n'apparait pas directement comment remédier à ce problème dans le modèle MIP. En effet la variance est un critère non linéaire et ne peut être modéliser facilement dans une approche MIP. De plus, une approche de PC peut exploiter directement l'aspect combinatoire de la structure du bin-packing et des contraintes additionnelles tandis que le MIP possède une relaxation linéaire généralement assez mauvaise pour les problèmes de bin-packing. Finalement, le modèle MIP n'évite pas certaines symétries du problème : pour une solution donnée, les infirmières sont complètement interchangeables. Dans la suite nous rappelons les contraintes d'équilibre existantes en PC avant d'introduire les modèles Comet de PC pour résoudre le problème.

FIGURE 1 – Comparaison entre deux solutions sur une instance à 6 infirmières, 14 enfants et 2 zones. La solution de gauche est obtenue en minimisant le critère `range-sum`. La solution de droite est obtenue en minimisant la variance des charges de travail.

4 Les contraintes d'équilibre en PC

Les contraintes d'équilibre apparaissent dans de nombreuses applications réelles, la plupart du temps pour exprimer une répartition équitable d'objets ou du travail. Par exemple, Simonis [15] suggère d'utiliser une contrainte globale pour équilibrer la distribution des équipes dans les problèmes de rostering. Pesant propose dans [7] d'utiliser une contrainte d'équilibre pour une allocation équitable des horaires individuels.

Deux contraintes globales et leurs propagateurs ont été proposés en programmation par contrainte pour optimiser l'équilibre : `spread` [6, 11], qui contraint la variance et la moyenne d'un ensemble de variables, et `deviation` [12, 13], qui contraint la moyenne et l'écart absolu moyen à la moyenne d'un ensemble de variables. On dit aussi que `spread` et `deviation` contraignent respectivement les normes L_2 et L_1 d'un ensemble de variables $X_1..X_n$ par rapport à leur moyenne ($s = \sum_{i\in[1..n]} X_i$), i.e.,

- $L_1 : \sum_{i\in[1..n]} |X_i - s/n|$;
- $L_2 : \sum_{i\in[1..n]} (X_i - s/n)^2$.

Ces deux critères ne sont pas équivalents : Minimiser L_1 ou L_2 ne conduit pas aux mêmes solutions et il n'est pas toujours évident de choisir l'un plutôt que l'autre. Ce choix entre L_1 et L_2 est récurrent et ne date pas d'aujourd'hui (voir par exemple [3]). Pour cette application, nous utilisons le critère L_2 et sa contrainte `spread` car L_2 est plus sensible aux points extrêmes que nous voulons absolument éviter dans cette application.

Nous utilisons les définitions et notations suivantes pour décrire la sémantique de la contrainte `spread` et de ses propagateurs.

Definition 1 *Soit X une variable à domaine fini (discrète). Le domaine de X est un ensemble de valeurs entières ordonnées pouvant être assignées à X dénoté $Dom(X)$. La valeur minimum (resp. maximum) du domaine est dénotée par $X^{\min} = \min(Dom(X))$ (resp. $X^{\max} = \max(Dom(X))$. Un intervalle entier aux bornes entières a et b est dénoté $[a..b] \subseteq \mathbb{Z}$, alors que nous dénotons $[a, b] \subseteq \mathbb{Q}$ l'intervalle rationnel. Une affectation des variables $\boldsymbol{X} = [X_1, X_2, ..., X_n]$ est dénoté par le tuple $\mathbf{x}$ et la ième entrée de ce tuple par $\mathbf{x}[i]$. L'intervalle domaine étendu rationnel de X_i est $I^{\mathbb{Q}}_D(X_i) = [X_i^{\min}, X_i^{\max}]$ et son intervalle domaine étendu entier est $I^{\mathbb{Z}}_D(X_i) = [X_i^{\min} .. X_i^{\max}]$.*

Nous définissons maintenant la contrainte `spread` avec moyenne fixe.

Definition 2 *Etant donné les variables à domaine fini $\boldsymbol{X} = (X_1, X_2, ..., X_n)$, une valeur entière s et une*

variable à domaine fini Δ, $\texttt{spread}(\boldsymbol{X}, s, \Delta)$ *est satisfaite si et seulement si*

$$\sum_{i\in[1..n]} X_i = s \quad \text{et} \quad \Delta \geq n \cdot \sum_{i\in[1..n]} |X_i - s/n|^2.$$

Observons également

$$n \cdot \sum_{i\in[1..n]} |X_i - s/n|^2 = n \cdot \sum_{i\in[1..n]} X_i^2 - s^2. \qquad (1)$$

Comme s est entier, cette quantité est entière. C'est la raison pour laquelle il est plus facile de travailler avec $n \cdot \sum_{i\in[1..n]} X_i^2 - s^2$ plutôt que $\sum_{i\in[1..n]} |X_i - s/n|^2$.

Exemple 1 $\mathbf{x} = (4, 6, 2, 5)$ *est une solution de* $\texttt{spread}([X_1, X_2, X_3, X_4], s = 17, \Delta = 40)$ *mais* $\mathbf{x} = (3, 6, 2, 6) \notin \texttt{spread}([X_1, X_2, X_3, X_4], s = 17, \Delta = 40)$ *car* $4 \cdot (3^2 + 6^2 + 2^2 + 6^2) - 17^2 = 51 > 40$.

L'algorithme de filtrage pour `spread` réalise la $\mathbb{Z}$-consistance aux bornes :

Definition 3 (Consistance aux bornes) *Une contrainte* $C(X_1, ..., X_n)$ $(n > 1)$ *est* $\mathbb{Q}$*-consistante aux bornes (resp.* $\mathbb{Z}$*-consistante aux bornes) par rapport aux domaines* $Dom(X_i)$ *si pour tout* $i \in \{1, ..., n\}$ *et chaque valeur* $v_i \in \{X_i^{\min}, X_i^{\max}\}$*, il existe les valeurs* $v_j \in I_D^{\mathbb{Q}}(X_j)$ *(resp.* $v_j \in I_D^{\mathbb{Z}}(X_j)$*) pour tout* $j \in \{1, ..., n\} - i$ *telles que* $(v_1, ..., v_n) \in C$.

Les propagateurs décrits dans [6, 11] réalisent la $\mathbb{Q}$-consistance aux bornes, ce qui signifie qu'ils considères que les variables peuvent être assignées à des nombres rationnels. Les propagateurs implémentés dans Comet réalisent la plus forte $\mathbb{Z}$-consistance aux bornes en adaptant les algorithmes de [6, 11]. En particulier pour réaliser la $\mathbb{Z}$-consistance aux bornes, les propagateurs de `spread` calculent $\underline{\Delta}^{\mathbb{Z}}$ pour filtrer $\Delta^{\min}$, $\overline{X}_i^{\mathbb{Z}}$ et $\underline{X}_i^{\mathbb{Z}}$ pour filtrer $X_i^{\max}$ and $X_i^{\min}$:

$$\underline{\Delta}^{\mathbb{Z}} = \min_{\mathbf{x}}\{n \cdot \sum_{i\in[1..n]} (\mathbf{x}[i] - s/n)^2 \text{ s.t. } \sum_{i\in[1..n]} \mathbf{x}[i] = s \quad (2)$$
$$\text{et } \forall i \in [1..n] : \mathbf{x}[i] \in I_D^{\mathbb{Z}}(X_i)\}$$

$$\overline{X}_i^{\mathbb{Z}} = \max_{\mathbf{x}}\{\mathbf{x}[i] \text{ s.t. } n \cdot \sum_{j\in[1..n]} (x[j] - s/n)^2 \leq \Delta^{\max} \quad (3)$$
$$\text{et } \sum_{j\in[1..n]} \mathbf{x}[j] = s \text{ et } \forall j : x[j] \in I_D^{\mathbb{Z}}(X_j)\}.$$

Le filtrage de Δ est implémenté dans le système Comet $O(n \cdot log(n))$ et celui de $\mathbf{X}$ en $O(n^2)$ [2, 10].

5 Un modèle simple de PC.

Nous présentons maintenant un modèle de PC qui adresse les problèmes du modèle MIP que nous avons relevés.

Le modèle de PC. Soit m le nombre d'infirmières, n le nombre de patients, et a_i l'acuité du patient i. L'ensemble des patients dans la zone k est dénoté $\mathcal{P}_k$ et $[\mathcal{P}_1, ..., \mathcal{P}_p]$ forme une partition de $\{1, ..., n\}$. Pour chaque patient i, nous introduisons une variable de décision $N_i \in [1..n]$ représentant l'infirmière qui s'occupe de lui. La charge de travail de l'infirmière j est représentée par la variable $W_j \in [0..acuity^{\max}]$. L'objectif et les contraintes sont modélisées comme suit.

- L'objectif, i.e., la minimisation de la norme L_2, est exprimé par la contrainte `spread` liant les variables de charges de travail $[W_1, ..., W_m]$, l'acuité totale, et la variable représentant la variance de l'acuité : `spread(`$[W_1, ..., W_m]$`,totalAcuity,spreadAcuity)`. Notons que `spreadAcuity` est la variable qu'il faut minimiser.
- Pour exprimer le fait que les infirmières ne peuvent avoir une acuité totale supérieure à $acuity^{\max}$, nous lions les variables N_i, W_j, et les acuités des patients avec une contrainte globale de multiknapsack [14] : `multiknapsack(`$[N_1, ..., N_n], [a_1, ..., a_n], [W_1, ..., W_m]$`)`.
- Pour modéliser le fait qu'une infirmière ne peut s'occuper de plus de $children^{\max}$ enfants, nous utilisons une contrainte globale de cardinalité [8] : `cardinality(`$1, [N_1, ..., N_n], children^{\max}$`)`.
- La contrainte qu'une infirmière ne peut travailler que dans une seule zone est modélisée avec une contrainte disant que toutes les paires de tableaux de variables ont des valeurs disjointes : `pairwiseDisjoint(`$[Z_1, ..., Z_p]$`)`, où Z_k est le tableau de variables contenant les variables N_i associées à la zone k.

Le programme Comet Le modèle Comet est donné dans le Listing 1. Les lignes 1–3 déclarent les variables de décision. La ligne 4 déclare les tableaux pour les zones qui sont remplis aux lignes 5–7. La fonction objectif est spécifiée aux lignes 8–9 et 11. Les lignes 12–14 sont les contraintes du problème. La contrainte `pairwiseDisjoint` introduit des variables ensemble représentant l'ensemble des infirmières travaillant dans une zone particulière $NS_k = \bigcup_{i\in\mathcal{P}_k} N_i$. L'ensemble NS_k est maintenu avec une contrainte globale `unionOf`. Ensuite, l'intersection vide de toutes les paires d'ensembles se fait avec une contrainte globale d'ensembles disjoints. Comet utilise une remormula-

tion de cette contrainte avec une contrainte de cardinalité comme expliqué dans [9, 1].

La recherche est implémentée dans le bloc `using` aux lignes 16–24. La recherche casse dynamiquement les symétries de valeurs induites par l'interchangeabilité des infirmières. Le patient ayant la plus grande acuité est sélectionné d'abord à la ligne 17. Ensuite la recherche tente d'assigner ce patient en commençant d'abord par les infirmières ayant la plus petite charge de travail (lignes 19–22). Le cassage des symétries est implémenté en considérant les infirmières déjà assignées et au plus une infirmière additionnelle n'ayant aucun patient (une technique similaire est utilisée pour résoudre le steel mill slab problem dans [4]). La valeur `mn` est l'indice maximum d'une infirmière déjà assignée à un patient. L'instruction `tryall` considère tous les indices d'infirmières jusqu'à `mn+1` (l'infirmière `mn+1` n'ayant pas de patient).

TABLE 1 – Résultats sur des instances à deux zones et minimisation du critère L_2 avec `spread`.

m	n	#fails	time(s)	avg workload	sd. workload
11	28	511095	170.2	86.09	2.64
11	29	1126480	302.0	80.27	1.76
10	26	104931	24.7	76.50	2.29
12	30	259147	136.5	83.42	1.93
10	28	2990450	1138.5	91.80	6.84
10	26	779969	206.9	88.40	2.29
12	29	555243	198.2	80.08	2.72
10	27	931858	343.9	90.60	5.33
10	25	1616689	434.5	82.70	7.32
8	22	4160	1.2	87.50	3.12

Résultats expérimentaux Pour la première expérience, nous avons généré 10 instances avec 2 zones, telles que les instances réelles étudiées dans [5]. Ces instances ont de l'ordre de 10–15 infirmières, 20–30 enfants, et ne peuvent être résolue avec le modèle MIP. Toutes les instances ont pu être résolues de manière optimale par notre modèle COMET en moins de 30 minutes (contrainte de temps spécifiée par l'hôpital dans [5]). La Table 1 montre les résultats expérimentaux. Tous les résultats utilisent COMET 1.1 [2] sur un Intel 2.4 GHz Core Duo avec 4GB sous MacOS 10.5.6.

6 Un modèle de PC en deux temps

Le modèle de PC basique peut résoudre des instances à deux zones mais a de grandes difficultés pour trois zones ou plus. Nous montrons comment simplifier la résolution par une approche en deux temps qui calcule d'abord le nombre d'infirmières assignées dans chaque zone et dans ensuite assigne les patients aux infirmières. Cette approche simplifie résolution en

1. supprimant le degré de flexibilité du nombre d'infirmières dans chaque zone.
2. supprimant la nécessité d'une contrainte d'ensembles disjoints puisque les ensembles d'infirmières pouvant être assignés à chaque patient sont précalculés.

Une Relaxation La première étape, c'est à dire déterminer le nombre d'infirmières dans chaque zone, est très importante. En effet si ces choix ne sont pas les bons, la solution finale peut être fortement sous optimale. Il est clair que le nombre d'infirmières assigné à chaque zone aura un grand impact sur la qualité de l'équilibre des charges de travail. Apres avoir visionné quelques solutions optimales, nous avons constaté que les temps de travail intra zone étaient trés équilibrés (quasiment les mêmes). Cela nous a inspiré la résolution d'une relaxation du problème pour découvrir une bonne répartition des infirmières entre les zones. La relaxation permet que l'acuité d'un enfant soit répartie sur plusieurs infirmières de la zone de l'enfant (relaxation continue de l'acuité). Etant donné que l'acuité peut être divisée, le problème relâché aura une solution optimale où toutes les infirmières d'une zone ont exactement la même charge de travail $\frac{A_k}{x_k}$, i.e., l'acuité totale $A_k = \sum_{i \in \mathcal{P}_k} a_i$ de la zone k divisé par le nombre d'infirmières x_k de la zone k. Cela est illustré sur la Figure 2 pour une relaxation d'un problème à deux zones. Le Lemme 1 justifie pourquoi la solution optimale de la relaxation prend une telle configuration. Intuitivement, tant que deux charges de travail peuvent être rendues plus similaires, le critère L_2 peut être diminué.

Lemme 1 *Etant donné m variables $[W_1, \ldots, W_m]$ avec une somme $s = \sum_{i=1}^{m} W_i$, le critère L_2 peut être amélioré si deux d'entre elles peuvent être rapprochées.*

Preuve 1 *Soit W_i et W_j les deux variables pouvant être rapprochées et considérons sans perte de généralité que $W_i > W_j$. Les variables après modification sont respectivement W'_i et W'_j. Si W_i et W_j sont rapprochées, cela signifie que $W'_i - W'_j < W_i - W_j$. Comme la somme est fixée nous avons $W'_i + W'_j = W_i + W_j$. Donc $W_i - W'_i = W'_j - W_j$ et il existe δ avec $\frac{(W_i - W_j)}{2} \geq \delta > 0$ tel que $W_i - W'_i = \delta = W'_j - W_j$. Nous avons $W'_i = W_i - \delta$ et $W'_j = W_j + \delta$. La somme des écarts quadratiques de départ avec la formule (1) est $\Delta = m \cdot \sum_{i=1}^{m} (W_i)^2 - s^2$. Avec W'_i et W'_j elle devient $\Delta' = m \cdot (\sum_{k \neq i,j} (W_k)^2 + (W_i - \delta)^2 + (W_j + \delta)^2) - s^2 = \Delta - 2\delta \cdot (W_i - W_j - \delta)$. Finalement comme $(W_i - W_j - \delta > 0)$, nous avons $\Delta' < \Delta$.*

Listing 1 – Modèle COMET pour l'affectation Patients-Infirmières

```
1  var<CP>{int} N[patients](cp,nurses);
2  var<CP>{int} W[nurses](cp,1..MaxAcuity);
3  var<CP>{int} spreadAcuity(cp,0..System.getMAXINT());
4  var<CP>{int}[] Z[zones];
5  int k = 1;
6  forall(i in zones,j in 1..nbPatientsInZone[i])
7      Z[i][j] = N[k++];
8  minimize<cp>
9     spreadAcuity
10 subject to {
11    cp.post(spread(W,sum(p in patients) acuity[p],spreadAcuity));
12    cp.post(multiknapsack(N,acuity,W));
13    cp.post(cardinality(minNbPatients,N,maxNbPatients));
14    cp.post(pairwiseDisjoint(Z));
15 }
16 using {
17   forall(p in patients: !N[p].bound()) by (−acuity[p],N[p].getSize()) {
18     int mn = max(0,maxBound(N));
19     tryall<cp>(n in nurses: n <= mn + 1) by (W[n].getMin())
20         cp.label(N[p],n);
21     onFailure
22         cp.diff(N[p],n);
23   }
24 }
```

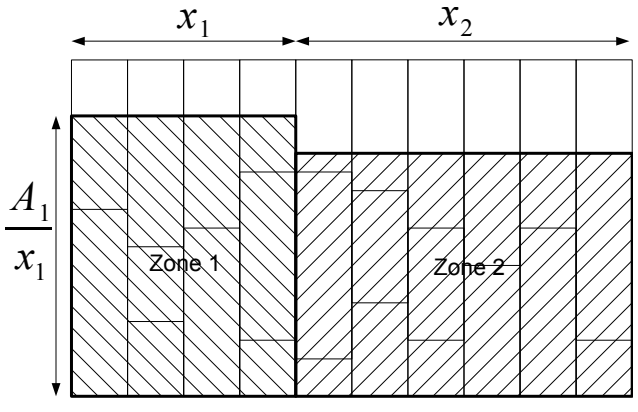


FIGURE 2 – Illustration d'une solution de la relaxation résolue pour trouver le nombre d'infirmières de chaque zone.

Le Lemme 1 prouve que la solution optimale du problème relâché comporte la même charge de travail pour toutes les infirmières d'une même zone. Par conséquent, la formulation mathématique du problème relâché est la suivante :

$$\min \quad \sum_{k=1}^{p} x_k \cdot \left(\frac{A_k}{x_k} - \sum_{j=1}^{p} \frac{A_j}{m} \right)^2 \tag{4}$$

$$s.t. \quad \sum_{k=1}^{p} x_k = m \tag{5}$$

$$x_k \in \mathbb{Z}_0^+ \tag{6}$$

La charge de travail des infirmières de la zone k est $\frac{A_k}{x_k}$ et la charge de travail moyenne est $\sum_{j=1}^{p} \frac{A_j}{m}$. Donc la contribution au critère L_2 pour les x_k infirmières de la zone k est $x_k \cdot \left(\frac{A_k}{x_k} - \sum_{j=1}^{p} \frac{A_j}{m} \right)^2$.

Résolution de la relaxation Dans notre modèle de PC, nous approximons la relaxation en $O(p \cdot log(p))$. D'abord nous solutionnons la relaxation continue du problème, i.e., nous laissons tomber la contrainte d'intégrité (6). La solution de ce problème d'optimisation continue est $x_k = m \cdot \frac{A_k}{\sum_{j=1}^{p} A_j}$, ce qui corres-

pond à attribuer une même charge de travail moyenne $\sum_{j=1}^{p} \frac{A_j}{m}$ pour toutes les infirmières. Ensuite cette solution continue $x_k = m \cdot \frac{A_k}{\sum_{j=1}^{p} A_j}$ peut être transformée de manière vorace en une solution entière en suivant les étapes suivantes :

- En développant la formule de l'objectif (4), il apparait qu'il est équivalent de minimiser $\sum_{k=1}^{p} \frac{(A_k)^2}{x_k}$.
- La transformation en une solution entière commence par arrondir vers le haut le nombre d'infirmières dans chaque zone $x_k = \lceil m \cdot \frac{A_k}{\sum_{j=1}^{p} A_j} \rceil$. La conséquence étant que la contrainte (5) peut être violée et la fonction objectif peut avoir diminué.
- Ensuite, les $x_k > 1$ sont considérés pour être diminués d'une unité jusqu'à rétablir la contrainte (5). L'indice k du x_k suivant qui est diminué est $\text{argmin}_k\{\frac{A_k^2}{x_k - 1} - \frac{A_k^2}{x_k}\}$, i.e., la variable qui augmente le moins son terme correspondant dans la fonction objectif $\sum_{k=1}^{p} \frac{A_k^2}{x_k}$.

Nos résultats expérimentaux montre que l'approximation est optimale sur toutes les instances résolues par le premier modèle de PC.

Borne inférieure sur la variance Le précalcul du nombre d'infirmières assignées à chaque zone peut aussi servir à calculer une borne inférieure sur le critère L_2. A l'intérieur d'une zone, la charge de travail moyenne est $\mu_k = A_k/x_k$. Puisque l'acuité des patients est entière, nous pouvons obtenir une meilleure borne inférieure sur l'objectif (4) en considérant que la charge de travail d'une infirmière de la zone k est soit $\lfloor \mu_k \rfloor$ soit $\lceil \mu_k \rceil$. Cela est illustré sur la Figure 3. Puisque la charge totale de travail pour la zone k doit rester A_k, la répartition des charges de travail entre $\lfloor \mu_k \rfloor$ et $\lceil \mu_k \rceil$ est donnée respectivement par $\alpha_k = A_k + x_k \cdot (1 - \lceil \mu_k \rceil)$ and $\beta_k = x_k - \alpha_k$. La borne inférieure sur la variable de variance $\underline{\Delta}^{\mathbb{Z}}$ calculée à l'aide de la formule (1) est donc

$$m \cdot \sum_{k=1}^{p} (\alpha_k \cdot \lceil \mu_k \rceil^2 + \beta_k \cdot \lfloor \mu_k \rfloor^2) - (\sum_{k=1}^{p} A_k)^2. \quad (7)$$

Le modèle Comet Le modèle Comet en deux temps est donné au Listing 2 et il considère que les x_k ont déjà été calculés. Le modèle ne crée pas les N variables à la ligne 2 : ces variables seront créées au même moment que le tableau de la zone puisque leur domaine est restreint à un sous ensemble d'infirmières. Les lignes 6–12 créent les tableaux de variable des zones, la ligne 10 construisant le tableau de variable de la zone i. Notons que le domaine de

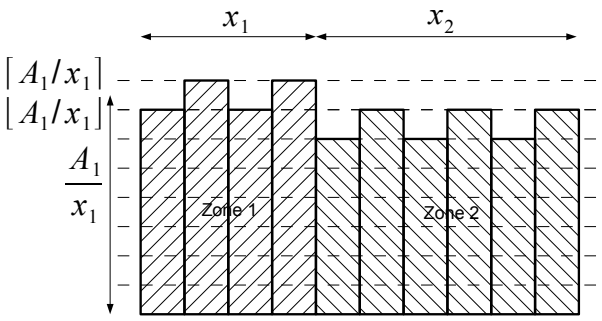


Figure 3 – Illustration de la borne inférieure sur L_2 en utilisant le précalcul du nombre d'infirmières dans chaque zone.

ces variables sont définis aux lignes 9 et 11 en utilisant le nombre d'infirmières assignées à chaque zone. Les lignes 13–15 assignent les variables de zone aux variables infirmière (le contraire du premier modèle car les variables de zone ont maintenant des domaines restreints). Les contraintes sont similaires mais il n'y a plus besoin de la contrainte `pairwiseDisjoint`. La recherche est implémentée aux lignes 23–34. Elle est un peu plus compliquée puisque les patients sont affectés une zone à la fois. Le cassage de symétries dynamiques est le même mais adapté à cette affectation par zone.

La Table 2 donne les résultats obtenus pour les même instances à deux zones que pour la Table 1 utilisant le précalcul du nombre d'infirmières allouées à chaque zone. La dernière collonne est la borne inférieure obtenue avec l'équation (7). Une première observation est que les temps de calculs sont fortement réduits. Ils n'excedent pas 10 secondes avec le nouveau modèle alors qu'ils pouvaient atteindre 1000 secondes pour les instances les plus difficiles avec le premier modèle. Le nombre d'infirmières trouvé dans la première étape est correct puisque les écarts types sont les mêmes que les valeurs optimales trouvées avec le premier modèle dans la Table 1. Il est également intéressant d'observer que la borne inférieure est raisonablement proche des valeurs optimales ce qui valide également l'approche.

Comme l'instance à deux zones peut maintenant être résolue aisément, nous avons essayé de résoudre des instances à trois zones. Les résultats sont présentés à la Table 3. Seules 6 instances sur 10 peuvent être résolues à l'optimum en moins de 30 minutes avec cette approche en deux temps.

Listing 2 – Modèle en deux temps pour l'assignation Patients-Infirmières.

```
1  Solver<CP> cp();
2  var<CP>{int} N[patients];
3  var<CP>{int} W[nurses](cp,1..MaxAcuity);
4  var<CP>{int} spreadAcuity(cp,0..System.getMAXINT());
5  var<CP>{int}[] Z[zones];
6  range nursesOfZone[zones];
7  int j=1;
8  forall(i in zones) {
9      nursesOfZone[i] = j..j+x[i]−1;
10     Z[i] = new var<CP>{int}[1..nbPatientsInZone[i]](cp,nursesOfZone[i]);
11     j += x[i];
12 }
13 int k = 1;
14 forall(i in zones,j in 1..x[i])
15         N[k++] = Z[i][j];
16 minimize<cp>
17     spreadAcuity
18 subject to {
19     cp.post(spread(W,sum(p in patients) acuity[p],spreadAcuity));
20     cp.post(multiknapsack(N,acuity,W));
21     cp.post(cardinality(minNbPatients,N,maxNbPatients));
22 }
23 using {
24   forall(i in zones){
25     forall(p in Z[i].rng(): !Z[i][p].bound()) by(−acuityByZone[i][p],Z[i][p].getSize()){
26         int shift = i==1? 0 : nursesOfZone[i−1].getUp();
27         int mn = max(0,maxBound(Z[i]))+shift;
28         tryall<cp>(n in nursesOfZone[i]: n <= mn + 1) by (W[n].getMin())
29             cp.label(Z[i][p],n);
30         onFailure
31             cp.diff(Z[i][p],n);
32     }
33   }
34 }
```

TABLE 2 – Résultats sur des instances à deux zones avec précalcul du nombre d'infirmières dans chaque zone.

m	n	#fails	time(s)	avg workload	sd. workload	lb. sd.
11	28	25385	4.5	86.09	2.64	2.23
11	29	4916	1.4	80.27	1.76	0.62
10	26	458	0.1	76.50	2.29	2.29
12	30	17558	6.7	83.42	1.93	1.19
10	28	29865	4.8	91.80	6.84	6.81
10	26	3705	1.0	88.40	2.29	1.43
12	29	6115	1.2	80.08	2.72	0.64
10	27	1109	0.4	90.60	5.33	5.22
10	25	3299	0.6	82.70	7.32	6.71
8	22	127	0.0	87.50	3.12	3.04

TABLE 3 – Résultats sur des instances à trois zones avec précalcul du nombre d'infirmières dans chaque zone.

sol	m	n	#fails	time(s)	avg wl.	sd. wl.	lb. sd.
1	15	42	19488	5.3	84.20	3.04	2.93
1	18	43	3619310	919.2	79.78	5.84	5.49
0	17	43	9023072	1800.0	81.41	4.75	3.45
1	17	42	483032	106.9	83.82	5.65	5.59
0	18	43	7124370	1800.0	81.00	7.11	4.94
1	14	38	590971	145.2	85.36	3.08	2.16
0	19	48	3786580	1800.0	87.42	3.18	2.30
1	16	44	3888210	839.8	84.88	6.70	6.39
0	19	49	5697272	1800.0	86.00	2.70	1.95
1	17	41	61250	17.3	82.18	3.40	3.07

7 Un modèle de PC en deux temps avec décomposition

L'approche précédente peut résoudre facilement les problèmes à deux zones mais présente des difficultés pour les instances à trois zones ou plus. Il paraît naturel de décomposer le problème en zone et d'équilibrer les charges de travail des infirmières dans chaque zone indépendamment plutôt que que d'équilibrer les charges de toutes les infirmières globalement. De manière intéressante, cette décomposition préserve l'optimalité, i.e., elle obtient la même solution pour le critère L_2 que l'approche en deux temps de la Section 6 pour un précalcul donné du nombre d'infirmières attribué à chaque zone. Autrement dit, étant donne un précalcul du nombre d'infirmières dans chaque zone, il est équivalent de minimiser L_2 entre toutes les infirmières en une fois ou de minimiser L_2 séparément dans chaque zone. Nous prouvons ce résultat formellement.

Lemme 2 *Minimiser* $n \cdot \sum_{i=1}^{x_k}(y_i - A_k/x_k)^2$ *tel que* $\sum_{i=1}^{x_k} y_i = A_k$ *est équivalent la minimisation de* $n \cdot \sum_{i=1}^{x_k}(y_i - (A_k/x_k + c))^2$ *tel que* $\sum_{i=1}^{x_k} y_i = A_k$.

TABLE 4 – Résultats sur des instances à trois zones avec précalcul du nombre d'infirmières dans chaque zone et décomposition par zone.

m	n	#fails	time(s)	avg workload	sd. workload	lb. sd.
15	42	203	0.1	84.20	3.04	2.93
18	43	608	0.1	79.78	5.84	5.49
17	43	8134	1.1	81.41	4.46	3.45
17	42	345	0.1	83.82	5.65	5.59
18	43	24994	3.2	81.00	5.77	4.94
14	38	151	0.0	85.36	3.08	2.16
19	48	3695	0.8	87.42	3.07	2.30
16	44	384	0.1	84.88	6.70	6.39
19	49	2056	0.4	86.00	2.49	1.95
17	41	776	0.2	82.18	3.40	3.07

Preuve 2 *Le premier objectif peut être reformulé au depart de la formule (1) comme* $x_k \cdot \sum_{i=1}^{x_k} y_i^2 - A_k^2$. *Le second peut être reformulé après quelques manipulations algébriques comme* $c^2 \cdot x_k^2 + x_k \cdot \sum_{i=1}^{x_k} y_i^2 - A_k^2$. *Comme ils ne diffèrent que par une constante, leur minimisation produit le même ensemble de solutions optimales.*

Corollaire 1 *Il est équivalent de minimiser* L_2 *entre toutes les infirmières en une fois ou de minimiser* L_2 *séparément dans chaque zone.*

Preuve 3 *Cela est une conséquence directe du Lemme 2. Si la minimisation de* L_2 *est faite globalement pour toutes les infirmières, le critère des moindres carrés* L_2 *est calculé par rapport à la charge moyenne de toutes les infirmière c'est à dire par rapport à* $\sum_{k=1}^{p} A_k/m$. *Cela correspond à un choix pour* c *dans le Lemme 2 égal à la différence entre la charge de travail moyenne dans la zone* k *et la charge globale moyenne :* $c = \sum_{k=1}^{p} A_k/m - A_k/x_k$.

Nous avons résolu à nouveau les instances à trois zones avec la méthode de la décomposition. Les résultats sont présentés sur la Table 4. Nous pouvons observer que comme attendu, les valeurs de la fonction objectif sont les mêmes pour toutes les instances qui ont été résolues de manière optimale dans la Table 3. Pour les autres, l'algorithme produit des solutions strictement meilleures. Le temps est également significativement moindre. La Figure 4 montre une visualisation COMET d'une solution à 15 zones avec 81 infirmières et 209 patients. Cette instances a pu être résolue en seulement 7 secondes et 10.938 échecs.

8 Conclusion

Cet article traite de l'affectation journalière des patients nouveaux nés aux infirmières dans un hôpital.

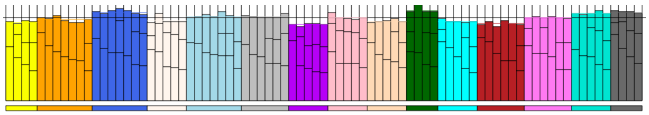


FIGURE 4 – Solution d'une instance à 15 zones.

L'objectif du problème est d'équilibrer la charge de travail des infirmières tout en satisfaisant certaines contraintes. Les travaux antécédents ont suggéré un modèle MIP pour résoudre ce problème qui avait deux limitations. Il ne permettait pas de résoudre des instances de grandes tailles et la fonction objectif ne modélise pas correctement le fait que l'on souhaite optimiser l'équilibre des charges de travail. Cet article présente une modèle de PC qui réalise correctement l'objectif d'équilibre et qui peut résoudre facilement des instances à deux zones. Pour permettre de résoudre des instances de grande taille, nous avons utilisé une décomposition du problème en deux temps : la première étape assigne les infirmière aux zones suivie de l'affectation des infirmières aux patients. La première étape est obtenue en solutionnant une relaxation du problème facile à résoudre. La seconde étape est résolue à l'aide d'une simplification du modèle de PC direct. Cette approche en deux temps améliore significativement les résultats sur les instances à deux zones et permet de résoudre des instances à trois zones. Nous montrons ensuite que les problèmes de chaque zone peuvent être résolus indépendamment sans perte de qualité. Le modèle de PC résultant résoud les problèmes à trois zones quasi instantanément et est très robuste quand le nombre de zones augmente. Par exemple, nous pouvons résoudre une problème à 15 zones, 81 infirmières et 209 patients en 7 seconds.

Il y a un certain nombre de problèmes intéressants à approfondir. Il serait intéressant d'étudier la qualité de l'approximation effectuée dans la première étape. Nos résultats expérimentaux indiquent que l'approximation est optimale sur toutes les instances testées néanmoins il est serait souhaitable d'avoir des garantie sur la qualité. Aussi nous pourrions envisager de résoudre la première étape de manière exacte. Nous devons envisager cet aspect algorithmique également. De plus, il serait intéressant d'étudier des problèmes où les infirmières ont diverses qualifications limittant leurs possibles affecations aux zones.

Références

[1] Christian Bessiere, Emmanuel Hebrard, Brahim Hnich, and Toby Walsh. Disjoint, partition and intersection constraints for set and multiset variables. In *Principles and Practice of Constraint Programming (CP 2004)*, pages 138–152, 2004.

[2] DYNADEC. Comet 1.1 release. *www.dynadec.com*, 2009.

[3] Stephen Gorard. Revisiting a 90-year-old debate : The advantages of the mean deviation. *British Journal of Educational Studies*, pages 417–439, 2005.

[4] P. Van Hentenryck and L. Michel. The steel mill slab design problem revisited. *CP'AI'OR-08, Paris, France*, 5015 :377–381, May 2008.

[5] C Mullinax and M Lawley. Assigning patients to nurses in neonatal intensive care. *Journal of the Operational Research Society*, 53 :25–35, 2002.

[6] G. Pesant and J.C. Régin. Spread : A balancing constraint based on statistics. *Lecture Notes in Computer Science*, 3709 :460–474, 2005.

[7] Gilles Pesant. Constraint-based rostering. *The 7th International Conference on the Practice and Theory of Automated Timetabling (PATAT 2008)*, 2008.

[8] J-C. Régin. Generalized arc consistency for global cardinality constraint. *AAAI-96*, pages 209–215, 1996.

[9] J.C. Régin. Habilitation à diriger des recherches (hdr) : Modelization and global constraints in constraint programming. *Université Nice*, 2004.

[10] P. Schaus. Balancing and bin-packing constraints in constraint programming. *PhD thesis, Université catholique de Louvain, INGI,*, 2009.

[11] P. Schaus, Y. Deville, P. Dupont, and J.C. Régin. Simplification and extension of spread. *3th Workshop on Constraint Propagation And Implementation*, 2006.

[12] P. Schaus, Y. Deville, P. Dupont, and J.C. Régin. The deviation constraint. *Proceedings of CP-AI-OR*, 4510 :269–284, 2007.

[13] Pierre Schaus, Yves Deville, and Pierre Dupont. Bound-consistent deviation constraint. *13th International Conference on Principles and Practice of Constraint Programming (CP 2007)*, 4741, September 2007.

[14] Paul Shaw. A constraint for bin packing. In *Principles and Practice of Constraint Programming (CP 2004)*, pages 648–662, 2004.

[15] Helmut Simonis. Models for global constraint applications. *Constraints*, 12 :63–92, March 2007.

Combiner Contraintes et Modèles pour le Traitement de Langage Contrôlé

Mathias Kleiner **Patrick Albert** **Jean Bézivin**

INRIA*, Centre Rennes - Bretagne Atlantique, Ecole des Mines de Nantes,
4 rue Alfred Kastler, 44307 Nantes, France
ILOG S.A, 9 rue de Verdun, 94253 Gentilly, France

mathias.kleiner@inria.fr

Résumé

Les schémas conceptuels (SC) sont des éléments centraux des systèmes d'information. Dans un processus d'administration, un problème récurrent et difficile est de permettre aux décideurs de directement définir et maintenir leurs schémas à l'aide d'un langage pseudo-naturel. Semantics for Business Vocabulary and Rules (SBVR), une spécification récemment publiée, offre une syntaxe abstraite permettant d'exprimer un SC et une syntaxe concrète à base d'Anglais structuré. Dans cet article, nous proposons une méthode originale pour extraire un modèle SBVR à partir d'un texte anglais puis le transformer en un diagramme de classe UML. Nous décrivons, dans un cadre d'ingénierie des modèles, une combinaison de programmation par contraintes orientée objet et de transformation de modèles. En plus des résultats théoriques, des expérimentations préliminaires sont fournies sur un exemple concret.

1 Introduction

Les schémas conceptuels (SC) sont largement utilisés dans l'industrie pour représenter de manière formelle les connaissances d'un système d'information. Un SC est souvent l'élément central sur lequel repose un ensemble d'opérations : validation, simulation, génération de cas d'utilisation, etc. La combinaison UML/OCL est le standard *de facto* pour la spécification de SC. Cependant la modélisation, la maintenance et l'évolution d'un SC requiert actuellement des compétences techniques importantes. Une approche récente en ingénierie logicielle cherche à faciliter ce processus en permettant aux décideurs d'exprimer leurs besoins en langage naturel pour ensuite les transformer dans une représentation formelle.

Dans le contexte commercial, l'Object Management Group (OMG) a récemment publié la spécification SBVR (Semantics for Business Vocabulary and Rules). SBVR offre un meta-modèle des concepts et des faits qui peut être utilisé pour définir un SC. La spécification propose également une syntaxe concrète sous la forme d'Anglais structuré. Cependant, l'analyse de langage naturel pour l'obtention d'un modèle SBVR est un problème difficile. En particulier, l'utilisation de grammaires *dépendantes du contexte* pour construire des langages contrôlés (structurés et spécifiques à un domaine) engendre la possibilité d'avoir des interprétations différentes d'une même phrase. Cette propriété disqualifie les algorithmes déterministes, comme les outils existant de l'ingénierie des modèles (IDM) (ATL[12], QVT[2]) et la plupart des *parseurs* de langages contrôlés (ACE[20]). Dans ces conditions, le problème devient combinatoire et requiert de *rechercher* des solutions à l'aide de techniques avancées comme celles utilisées en intelligence artificielle.

La principale contribution de cet article est une traduction automatique d'Anglais contrôlé en un modèle SBVR et un modèle UML du SC décrit, permettant de construire des langages flexibles (éventuellement spécifiques à un domaine). L'originalité de notre approche est qu'elle combine, dans un cadre d'IDM, les techniques de programmation par contraintes (PPC) et les outils de transformation de modèles. Elle est principalement composée de trois opérations. La première est une analyse syntaxique et grammaticale du texte, directement liée au domaine difficile du traitement du

*Team AtlanMod

langage : nous décrivons un parseur basé sur une approche de la PPC orientée objet appelée configuration [13]. La seconde tâche est une transformation du modèle résultant en un modèle SBVR. La troisième tâche est une transformation de ce modèle SBVR en un modèle UML du SC. Parallèlement, l'intégration de la PPC dans l'IDM en tant qu'outil avancé de transformation de modèles est une contribution importante et innovante de ce travail.

1.1 Plan de l'article

La Section 2 introduit brièvement les technologies employées dans notre approche. Nous présentons également une vue d'ensemble du processus et un exemple concret. La Section 3 introduit le parseur de langage. La Section 4 montre comment le modèle résultant est transformé en un modèle SBVR. La Section 5 propose une transformation de SBVR vers UML. L'implémentation et les expérimentations sont présentées en Section 6. Enfin, nous discutons de l'état de l'art en Section 7.

2 Contexte de travail

2.1 Brève introduction à SBVR

SBVR est un standard de l'OMG[3] dont le but est d'être une base pour la description d'activités commerciales en langage naturel. C'est une tentative pour combler l'écart entre les utilisateurs finaux et l'architecture logicielle, en permettant aux non-spécialistes de paramétrer et faire évoluer la logique commerciale de leurs applications.

SBVR standardise un ensemble de concepts permettant de définir des langages contrôlés (langages déclaratifs dont la grammaire et le lexique sont limités pour éliminer une partie de l'ambiguïté). Voir [20] pour un article historique, et plus récemment [15]). Les systèmes de règles métier comme ILOG JRules ou Drools sont des langages contrôlés populaires utilisés pour modéliser explicitement la logique commerciale dans un nombre croissant d'applications.

Nouns ne décrirons pas SBVR de manière exhaustive dans cet article. Cependant les figures 1 et 2 illustrent la sophistication des meta-modèles et l'approche qui consiste à séparer les formulations logiques des concepts et des faits.

2.2 Brève introduction à l'IDM et la transformation de modèles

L'ingénierie des modèles est un domaine de recherche émergeant qui considère les artefacts logiciels principaux comme des graphes typés. Cela provient du besoin industriel d'avoir une organisation régulière et homogène où les différents aspects d'une architecture logicielle peuvent être séparés ou combinés.

En IDM, les modèles sont le concept unificateur. La communauté utilise les concepts de modèle terminal, méta-modèle et méta-méta-modèle. Un modèle terminal est la représentation d'un système. Il capture certaines de ses caractéristiques et fournit des connaissances. Les outils de l'IDM agissent sur des modèles terminaux exprimés dans des langages précis de modélisation. La syntaxe abstraite d'un langage de modélisation est appelée méta-modèle quand elle est exprimée elle-même comme un modèle. La relation entre un modèle et le méta-modèle de son langage est appelée *conformsTo*. Les méta-modèles sont eux-mêmes exprimés dans un langage de modélisation dont les bases conceptuelles sont capturées par un modèle auto-descriptif appelé méta-méta-modèle. L'architecture résultante est composée de trois niveaux appelés M1, M2, M3. Une définition formelle de ces concepts peut-être trouvée dans [11]. Les principes de l'IDM peuvent être implémentés dans différents standards. Par exemple, l'OMG propose un méta-méta-modèle standard appelé Meta Object Facility (MOF).

L'automatisation de l'IDM est principalement assurée par des méthodes de transformation. La production d'un modèle Mb à partir d'un modèle Ma par une transformation Mt est appelée transformation de modèle. Quand le méta-modèle source est identique à la cible (MMa = MMb), la transformation est dite endogène. Sinon, la transformation est exogène. Dans ce travail nous utilisons ATL (AtlanMod Transformation Language), un langage [12] de type QVT permettant une expression déclarative de la transformation à travers un ensemble de règles.

2.3 Brève introduction à la configuration

Configurer consiste à composer un système complexe à partir de composants génériques[13]. Les composants, aussi appélés objets ou éléments de modèle dans la suite, sont définis par leur type, attributs et relations mutuelles. Les systèmes acceptables, spécifiés par des contraintes, sont de plus restreints par la requête : un ensemble de besoins spécifiques, représentés par un fragment du système souhaité (i.e des objets interconnectés). Du point de vue de la représentation des connaissances, la configuration peut-être assimilée à la recherche d'un graphe typé et attribué (la structure) obéissant aux restrictions d'un modèle objet sous contraintes. Du point de vue de l'IDM, le problème est de trouver un modèle fini conforme à un méta-modèle. Plus précisément, nous considérons le processus comme une transformation de modèle dans

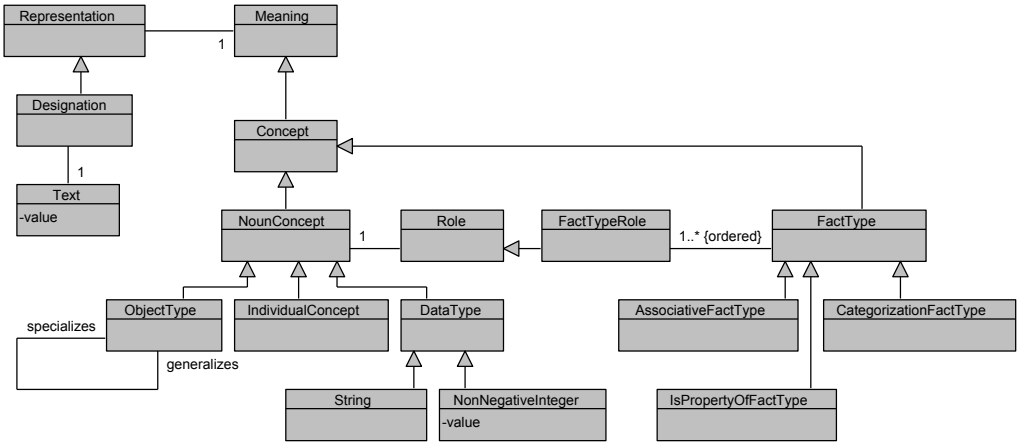


FIGURE 1 – Extrait du meta-modèle SBVR : concepts et faits

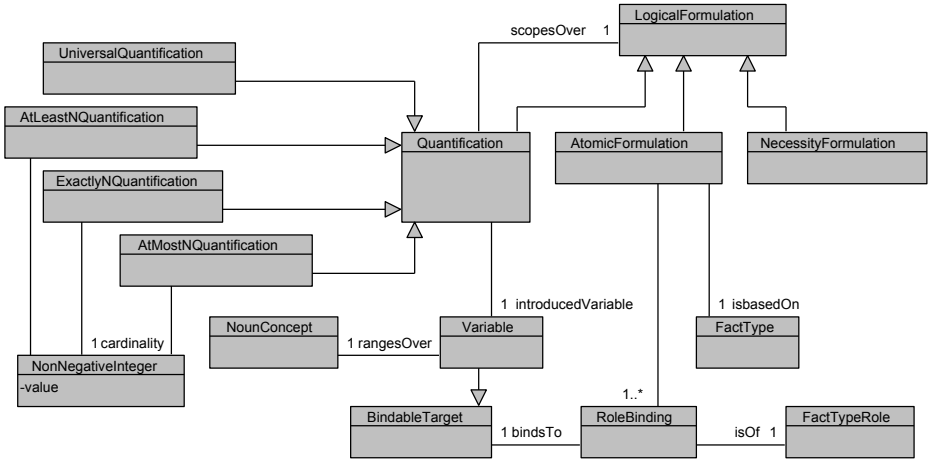


FIGURE 2 – Extrait du meta-modèle SBVR : formulations logiques

laquelle le modèle source est la requête et le modèle cible la solution. Le modèle de configuration joue alors le rôle de méta-modèle cible. Une version *relaxée* de ce méta-modèle définit le méta-modèle source. Elle est obtenue par le retrait de toutes les contraintes. Dans ces conditions, la requête (un modèle cible incomplet) est conforme au méta-modèle source et la configuration peut-être insérée comme une technique avancée de transformation de modèle : le configurateur *recherche* un modèle, en complétant la source et en y ajoutant tous les éléments de modèle nécéssaires pour que le résultat soit conforme aux contraintes du méta-modèle cible.

Plusieurs techniques ont été proposées pour traiter des problèmes de configuration : extensions des CSP (Constraint Satisfaction Problem) [18, 23, 19], approches à base de connaissances [22], programmation logique [21], approches orientées objet [17, 14]. La configuration est traditionnellement utilisée dans des applications industrielles comme la simulation de production. Plus récemment, les capacités expressives du formalisme ont montré son utilité pour la résolution de problèmes d'intelligence artificielle comme le traitement du langage [8]. Une introduction plus poussée à la configuration peut être trouvée dans [13].

Dans la suite nous proposons d'utiliser Ilog JConfigurator[14] pour l'analyse de langage contrôlé vers SBVR. Dans cette approche, un modèle de configuration (dans notre contexte, le méta-modèle cible) est bien défini par un ensemble de classes, relations et contraintes. Le langage UML/OCL peut être utilisé pour ces spécifications [7].

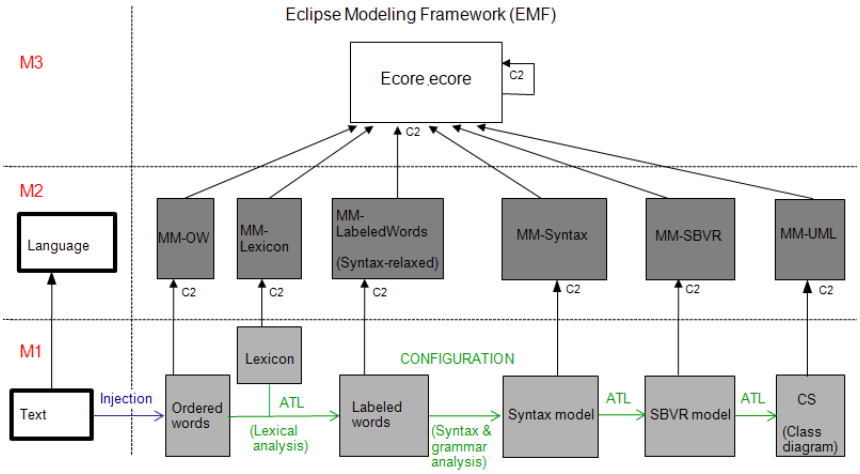


FIGURE 3 – Vue d'ensemble du processus

2.4 Vue d'ensemble du processus

La figure 3 montre l'ensemble du processus dans un cadre d'IDM. L'entrée est un texte Anglais, proche de la forme structurée proposée par le standard SBVR [3]. Le texte est injecté en modèle dont le méta-modèle est une simple annotation de la position des mots et des phrases dans le texte. Une transformation triviale utilise un lexique pour labeliser chaque mot par un ensemble de ses catégories syntaxiques possibles. Nous définissons ensuite un méta-modèle, appelé *Syntax*, pour lequel nous avons adapté les grammaires de configuration [8] à SBVR et le contexte de l'IDM. Le texte (en tant que mots labelisés) est fourni au configurateur grâce à la version relâchée de Syntax. Le résultat de cette analyse syntaxique et grammaticale est donc un modèle fini conforme au méta-modèle Syntax. Ce modèle est ensuite transformé avec ATL en un modèle conforme au méta-modèle SBVR grâce à un ensemble de règles utilisant les dépendances grammaticales générées. Ce modèle terminal SBVR peut ensuite être de nouveau transformé via ATL pour obtenir un modèle UML correspondant.

2.5 Exemple

Un exemple sera utilisé tout au long de cet article pour illustrer l'approche. Le texte considéré est composé de trois phrases définissant un SC :
(1) Each company sells at least one product.
(2) Each product is sold by exactly one company.
(3) A software is a product.
Cet exemple relativement simple contient pourtant des concepts non triviaux. Du point de vue du langage, nous utilisons des noms, des verbes à la forme passive ou active, ainsi que différents quantificateurs. Les phrases sont liées par le contexte car elles décrivent différents aspects des mêmes concepts. Du point de vue de la modélisation l'exemple décrit les notions de classes, d'héritage et de relations contraintes. Dans les sections qui suivent, nous montrerons l'application de chaque opération sur ces phrases.

3 Traitement de langue Anglaise contrôlée pour SBVR

Le traitement du langage naturel est un défi majeur de l'intelligence artificielle. Au vu de la difficulté du problème, de nombreux efforts ont été portés vers le domaine plus accessible des langage contrôlés [16], où des ambiguïtés sont retirées et le vocabulaire restreint. Parmi les approches existantes, [8] a montré que les grammaires de propriétés [4] peuvent être capturées dans un modèle de configuration afin de traiter un sous-ensemble de la langue française. Le parseur résultant n'hérite pas du comportement déterministe de la plupart des approches et est prévu pour être adapté à différentes grammaires. Nous avons modifié et étendu cette méthode pour l'Anglais et SBVR. Dans notre cadre d'IDM, le modèle de configuration est défini en tant que méta-modèle (Syntax).

3.1 Méta-modèle Syntax

Un fragment de ce méta-modèle (principalement les classes et les relations) est présenté dans la figure 4. Syntax capture trois types d'information à partir du texte : les catégories syntaxiques, les dépendances grammaticales et la sémantique SBVR.

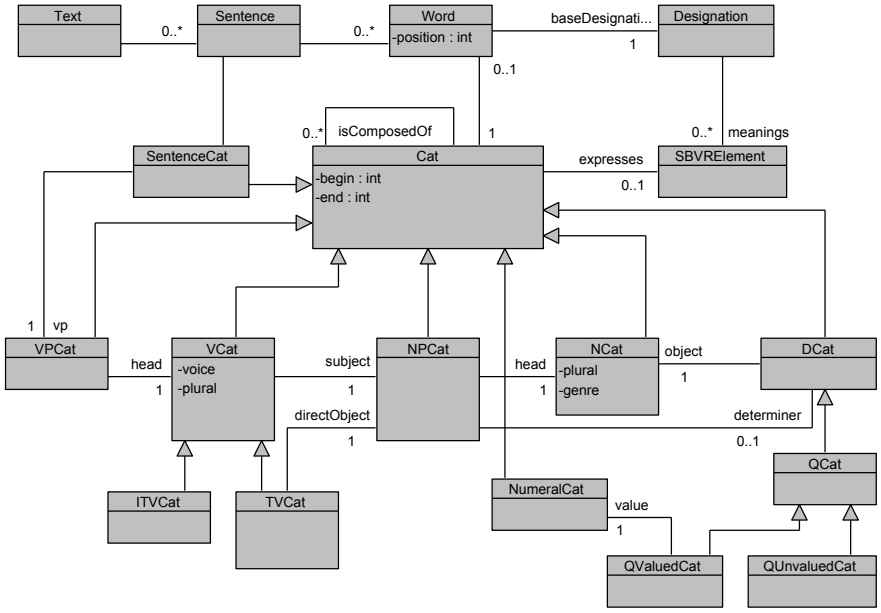


FIGURE 4 – Extrait du méta-modèle Syntax : syntaxe et grammaire

3.1.1 Analyse syntaxique

Afin d'obtenir un arbre syntaxique à partir d'une phrase, nous avons adapté le modèle des grammaires de propriété à l'Anglais. La classes principale est *Cat* et représente une catégorie syntaxique. Une catégorie est *terminale* quand elle est directement associée à un mot. Ces catégories incluent *NCat* (nom), *VCat* (verbe) ou *DCat* (déterminant). Ces catégories peuvent être spécialisées : un verbe est soit transitif (*TVCat*) ou intransitif (*ITVCat*). Les catégories possibles d'un mot sont obtenues grâce au lexique par la transformation précédente. Une catégorie est *non-terminale* quand elle est composée d'autres catégories. *SentenceCat* (phrase), *NPCat* (groupe nominal), *VPCat* (groupe verbal) sont les catégories non-terminales principales. Un ensemble de contraintes définit les catégorisations acceptables. Ces contraintes impliquent par exemple les constituants ou leur position relative dans une phrase. Voici quelques exemples de ces contraintes en OCL :

- *Chaque groupe verbal dont le verbe est transitif est composé d'au moins un groupe nominal.* Cette contrainte s'applique à la relation *isComposedOf* d'une catégorie :

```
context VPCat
inv: head.oclIsTypeOf(TVCat)
        implies isComposedOf->
          exists( elt.oclIsTypeOf(NPCat) )
```

- *Un groupe verbal est toujours précédé d'un groupe nominal.* Cette contrainte s'applique aux attributs *begin* et *end* des catégories (obtenus par la position des mots associés) :

```
context SentenceCat
inv: isComposedOf->exists(
        elt.oclIsTypeOf(NPCat)
        and elt.end < vp.begin )
```

3.1.2 Dépendances grammaticales

Nous avons étendu le modèle syntaxique pour faire apparaître explicitement les dépendances grammaticales en tant que relations entre les catégories. De la même manière, un ensemble de contraintes définit les constructions possibles. Voici quelques exemples de ces contraintes spécifiées en OCL :

- *Le sujet d'un verbe actif est placé avant le groupe verbal* :

```
context VPCat
inv: (head.voice = 'active')
        implies head.subject.end < begin
```

- *Un verbe et la tête de son sujet partagent le même pluriel* :

```
context VCat
inv: plural = subject.head.plural
```

3.1.3 Sémantique SBVR

Le meta-modèle est également enrichi par les principaux concepts de SBVR. Cette sémantique est assignée aux catégories grâce à la relation *expresses*. De nouveau, les contraintes gouvernent les affectations possibles. Quelques exemples :

– *Un verbe transitif exprime un* FactType :

```
context TVCat
inv: not expresses.oclIsUndefined()
        and expresses.oclIsKindOf(FactType)
```

– *La tête du sujet d'un verbe exprime soit un* ObjectType *soit un* IndividualConcept :

```
context VCat
inv: subject.head.expresses.
       oclIsKindOf(ObjectType)
     or subject.head.expresses.
       oclIsKindOf(IndividualConcept)
```

A propos de l'unicité des concepts SBVR Une difficulté majeure dans l'assignation de sémantique SBVR réside dans l'unicité des concepts. Plus précisément, le même concept SBVR peut être exprimé dans plusieurs phrases (voire la même). Considérons les deux premières phrases de notre exemple : les concepts "Company", "Product" et "To sell" sont exprimés plusieurs fois. Il faut éviter de dupliquer les éléments SBVR dans le modèle résultat. Un ensemble de contraintes force l'unicité de ces éléments sur la base d'hypothèses d'équivalence. Dans le cas d'éléments de la classe *NounConcept*, la désambiguation est faite sur la désignation des mots. Elle peut être formalisée en OCL de la manière suivante :

```
inv: NCat.allInstances()->
       forAll(n1,n2 : NCat |
       (n1.word.baseDesignation =
        n2.word.baseDesignation)
       = (n1.expresses = n2.expresses))
```

Puisque la désignation *canonique* est utilisée, différentes formes du même mot sont reconnues (i.e "products" et "product"). Le même principe est appliqué à d'autres éléments SBVR tels que les *FactType*.

3.2 Processus d'analyse et résultat

Comme expliqué précédemment, l'entrée du processus de configuration est un modèle d'une version relâchée du méta-modèle cible. Dans notre contexte, l'entrée est un ensemble d'objets inter-connectés de type *Text*, *Sentence*, *Word*, *Designation* et *Cat*. En effet, la transformation précédente, basée sur un lexique, a fourni les propriétés de chaque mot (pluriel, genre), leur désignation canonique ("has" a pour base "to have"), et les catégories syntaxiques candidates (le mot "one" peut être un nom, un nombre ou un adjectif).

Le résultat du processus de configuration est un (ou plusieurs) modèle(s) terminal(aux) satisfiant les contraintes, quand un tel modèle existe. La Figure 5 montre (un fragment) du modèle généré pour la phrase "Each company sells at least one product".
Il faut noter que ce processus n'est pas déterministe : à cause des ambiguïtés du langage, plusieurs solutions peuvent êtres valides pour une même requête. Par exemple, considérons la phrase "MyCode is a software". Sans information lexicale ou contexte, il n'est pas possible de décider si le *NounConcept* "MyCode" est un *ObjectType* (une spécialisation de "Software") ou un *IndividualConcept* (une instanciation de "Software"). Plutôt que de décider arbitrairement de restrictions dans le langage, notre méthode permet de générer toutes les solutions valides afin de les comparer, ou même d'optimiser le modèle cible sur la base de préférences. De ce point de vue, notre approche offre une flexibilité supérieure à la plupart des analyseurs.

4 Transformation du modèle syntaxique en un modèle SBVR

Le modèle produit par l'analyse exhibe la sémantique SBVR exprimée par des (groupes de) mots. Grâce à ces informations et les dépendances grammaticales entre les catégories, il est possible de construire un modèle SBVR complet du texte fourni. Cette opération est réalisée avec ATL[12], un outil de transformation de modèles dans lequel la mise en correspondance est exprimée par des règles déclaratives entre un méta-modèle source (ici, Syntax) et un méta-modèle cible (ici, SBVR). Nous ne présentons pas exhaustivement chaque règle de la transformation mais nous donnons ci-dessous ses principaux ressorts. [1]

4.1 Vue d'ensemble de la transformation

Une première mise en correspondance est évidente : chaque *SBVRElement* de Syntax a son équivalent dans SBVR et implique donc la création de cet élément cible. Cependant, les relations entre les éléments SBVR ne sont pas explicites dans le modèle source et peuvent nécessiter des éléments intermédiaires. Un ensemble de règles permet de les dériver à partir des relations grammaticales.

Par exemple, considérons la règle suivante qui génère un *AssociativeFactType* (binaire) et ses rôles à partir d'un verbe transitif, son sujet et son complément direct. Elle peut s'écrire de manière informelle : "Pour tout AssociativeFactType B exprimé par un verbe V du modèle source, créer un AssociativeFactType B', avec deux roles R1 et R2, où le concept de R1 est le concept cible du sujet de V et le concept de R2 est le concept cible du complément de V". La règle crée des éléments intermédiaires (les rôles) et les utilise pour mettre en relation les concepts SBVR. Certains de ces

1. Le code source et la documentation des transformations présentées dans cet article ont été soumis en tant que contribution Eclipse au projet ATL et sont disponibles sur http ://www.eclipse.org/m2m/atl/

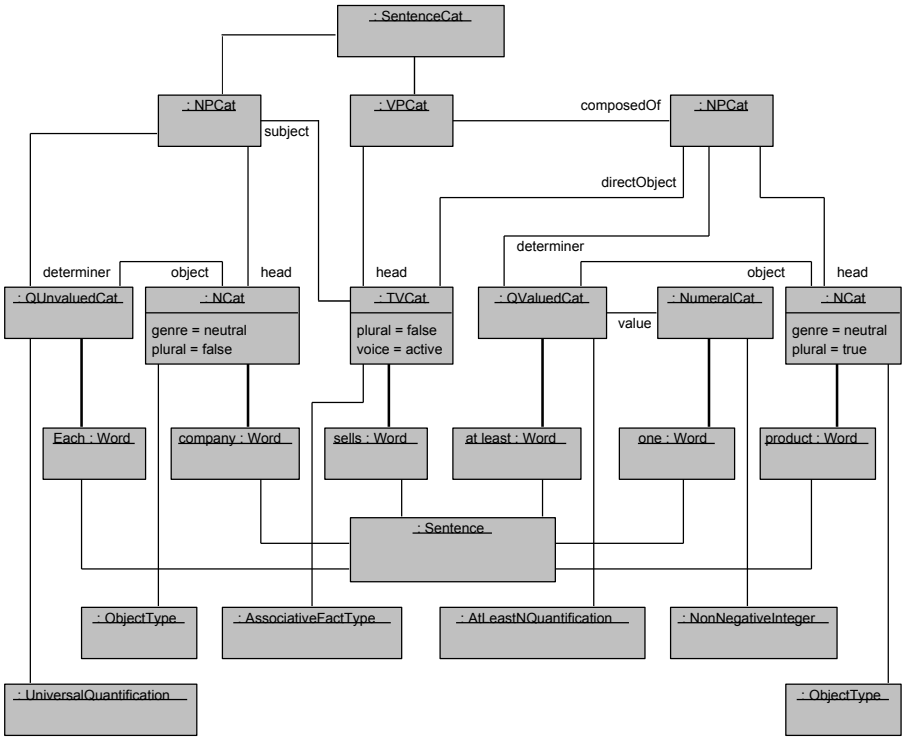


FIGURE 5 – Exemple : extrait d'un modèle terminal de Syntax

éléments (concepts cibles du sujet et du complément) étant eux-même crées par une autre règle.

La transformation permet également de créer des valeurs d'attributs à partir d'une source d'information d'un type différent. En effet, dans la première phrase de notre exemple, le mot "one" est associé à la catégorie *NumeralCat* et exprime un entier (*NonNegativeInteger*). La règle qui crée l'élément cible affectera la valeur 1 à l'attribut *value* de type *Integer*.

4.2 Processus de transformation et résultat

Une fois la transformation réalisée nous obtenons un modèle conforme à SBVR qui ne possède plus d'information syntaxique. La Figure 6 montre (un fragment) du modèle SBVR généré pour la première phrase de notre exemple.

5 Transformation du modèle SBVR en un modèle UML

Le modèle SBVR généré peut ensuite être transformé avec ATL en un modèle UML[**?**] correspondant au SC. De nouveau, nous ne présentons pas chaque règle mais les principales mises en correspondance.

5.1 Vue d'ensemble de la transformation

Quelques exemples sont présentés dans la Table 1 où la notation pointée est utilisée pour naviguer à travers les attributs et les relations. Ces correspondances sont assez naturelles : un type d'objet devient une classe, un type de fait associatif devient une association, une catégorisation désigne l'héritage (Generalization en UML), une propriété de type de fait se réfère à un attribut, un concept individuel devient une instance, etc. Les liens entre les concepts et les valeurs sont également explicites. Cependant, la plupart des règles ne réalisent pas une transformation unaire triviale. Par exemple, le concept SBVR *AtLeastNQuantification*. La propriété (*Property*) pour laquelle la valeur minimale (*lowerValue*) est affectée est celle obtenue par la transformation d'un fait (*AssociativeFactType*) cible de la relation *AtLeastNQuantification.scopesOver.isBasedOn*.

5.2 Processus de transformation et résultat

Une fois la transformation réalisée, nous obtenons une spécification UML du SC. La Figure 7 montre un modèle UML terminal pour notre exemple.

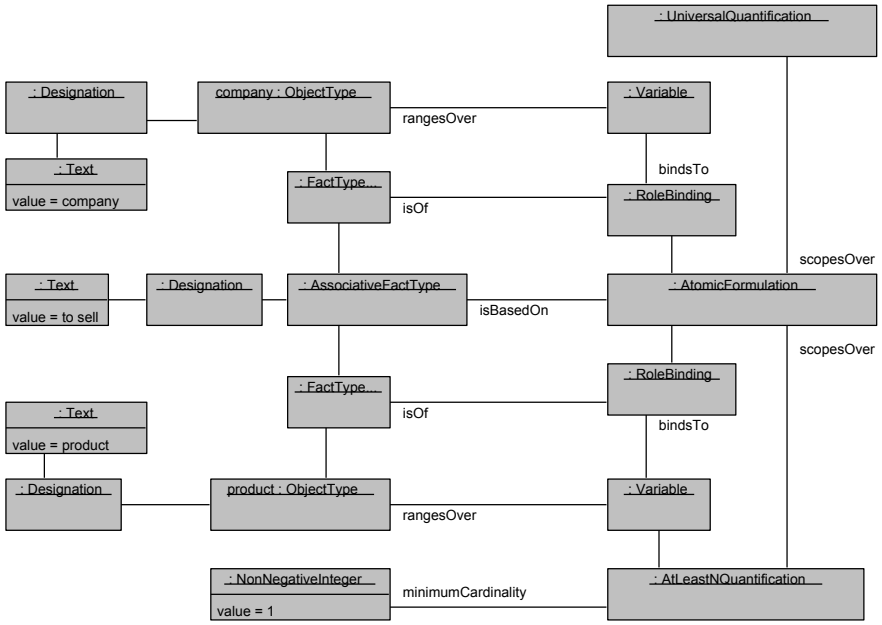


FIGURE 6 – Exemple : extrait d'un modèle terminal de SBVR

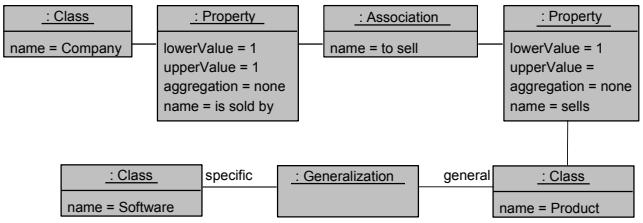


FIGURE 7 – Exemple : extrait du modèle UML

phrase(s)	*Temps*	*Vars*	*Contraintes*
(1)	0.26	527	1025
(2)	0.20	526	1022
(3)	0.19	475	885
(1)+(2)	1.82	973	2819
(1)+(2)+(3)	5.21	1328	5312

TABLE 2 – Expérimentations sur l'exemple (temps en secondes)

6 Implémentation et expérimentations

6.1 Implémentation

L'approche a été intégrée dans un cadre d'IDM basé sur Eclipse. Pour cela, chaque meta-modèle présenté a été défini grâce au langage de méta-modélisation KM3 [11], qui offre une conversion automatique vers le format ECore de l'EMF [1]. Ces méta-modèles ECore sont les sources et les cibles des transformations ATL proposées. Le configurateur (JConfigurator) a son propre langage. Le meta-modèle Syntax est donc défini également dans l'outil en tant que modèle de configuration. A l'execution, le modèle représentant la requête est extrait vers XML pour l'analyse, et la solution XML est injectée en un modèle de Syntax au format ECore. Ce modèle, au format XMI, est ensuite transmis aux transformations ATL.

6.2 Expérimentations

Nous présentons des expérimentations préliminaires sur l'exemple proposé, conduites sur un Intel Core2Duo 3Ghz - 3GB RAM. La Table 6.2 montre les résultats. Nous avons d'abord analysé chaque phrase séparément puis plusieurs phrases à la fois. Seuls les temps du configurateur sont affichés car les transformations ATL se déroulent en un temps négligeable (moins de 0.2 secondes).

L'analyse est efficace pour chaque phrase individuellement, mais le temps requis pour la recherche croit rapidement avec le texte complet. Ceci est dû à la taille du modèle source qui impacte directement l'espace de recherche du configurateur. D'un autre côté, les transformations ATL peuvent gérer de larges modèles. La séparation des tâches est donc positive pour la performance et pourrait être développée afin de réduire la

SBVR concept	UML Concept
ObjectType	Class
ObjectType.Designation.Text.value	Class.name
DataType	DataType
IndividualConcept	InstanceSpecification
AssociativeFactType	Association
AssociativeFactType.Designation.Text.value	Association.name
AssociativeFactType.FactTypeRole#1	Property (Association.memberEnd)
IsPropertyOfFactType.FactTypeRole#1.nounConcept	Property.classifier
CategorizationFactType	Generalization
CategorizationFactType.FactTypeRole#1	Generalization.general
AtLeastNQuantification.minimumCardinality.value	Property.lowerValue

Table 1 – Extrait du mapping entre les concepts SBVR et UML

complexité du modèle de configuration au minimum requis pour l'analyse syntaxique. Une autre alternative envisagée est d'analyser le texte de manière incrémentale, phrase par phrase, puis d'utiliser les capacités multi-source d'ATL pour unifier les modèles SBVR résultats. Il faut cependant noter que ce sont des expériementations préliminaires sur un problème connu pour être difficile. Aucune optimisation (heuristiques, propagation ou élimination de symétries) n'a été appliquée au moteur de configuration. Ces techniques sont fréquemment employées pour réduire les temps de calcul des problèmes sous contraintes. L'analyse réussie de notre exemple montre cependant la faisabilité de l'approche.

7 Travaux relatifs

[10] décrit la transformation inverse : à partir d'une description UML/OCL d'un SC, les auteurs montrent sa transformation en un modèle SBVR (via ATL), puis son paraphrasage en un texte Anglais structuré. La combinaison des deux approches est prometteuse. En effet, la conception d'un SC requiert généralement de nombreuses discussions entre décideurs et concepteurs, et la maintenance d'un SC amène de nombreuses évolutions. La combinaison pourrait permettre de traduire automatiquement les modifications d'une réprésentation vers une autre.
Des recherches précédentes ont été conduites sur l'utilisation des techniques de programmation par contraintes pour l'IDM, principalement sur l'animation de spécifications relationnelles ou la vérification de modèles. [5] transforme des spécifications UML/OCL en un problème CSP afin de vérifier la satisfaisabilité. [6] propose une méthode similaire, bien que le solveur (Alloy [9]) utilisé soit basé sur SAT. Les deux approches héritent des limitations du formalisme vers lequel les spécifications sont traduites, alors que le paradigme de la configuration est suffisamment expressif pour capturer directement la représentation sous forme de modèles. A notre connaissance, aucun travail existant n'a proposé l'utilisation de la recherche à base de contraintes comme outil de transformation pour l'IDM. En ce qui concerne le domaine de l'analyse de langages naturels contrôlés, notre approche diffère de la plupart des méthodes existantes (comme ACE[20]). En effet, ces parseurs n'acceptent pas de grammaire aimbigües, alors que nous pouvons paramétrer le niveau d'ambiguïtés acceptées. Ceci nous permet de mettre en balance la couverture du langage et l'efficacité computationnelle.

8 Conclusion et perspectives

Nous avons décrit une méthode qui permet d'analyser la spécification d'un SC, exprimée sous la forme de texte Anglais, et de la transformer un modèle de classe UML. Le standard de l'OMG SBVR est utilisé comme niveau intermédiaire de réprésentation. L'originalité de notre approche réside dans l'utilisation d'une technique avancée de programmation par contraintes, la configuration orientée-objet, en tant qu'outil de transformation de modèles intégré dans une architecture d'IDM. Des expérimentations préliminaires sont fournies pour vérifier la faisabilité de l'approche. De plus, l'analyseur présenté est flexible vis à vis de la couverture du langage et de l'ambiguïté acceptée, permettant ainsi de construire des langages contrôlés (spécifiques à un domaine) qui ne soient pas restreints à des grammaires libre de contexte. Ce travail offre plusieurs perspectives. Tout d'abord, les méta-modèles peuvent être étendus pour capturer un fragment plus important de la langue et de la sémantique SBVR. La génération de contraintes OCL, en plus d'UML, sera probablement nécessaire pour exprimer des sémantiques plus complexe. D'autres formalismes, tels qu'OWL ou

les systèmes à base de règles, peuvent être envisagés comme cibles finales du processus. Enfin, l'utilisation de la configuration peut être utile pour d'autres opérations complexes de l'IDM, dans lesquelles les transformations à base de règles déterministes ne sont pas suffisantes, et qui nécessitent de rechercher un modèle satisfaisant ou optimal parmi un ensemble de solutions potentielles.

Références

[1] Emf ecore : http ://www.eclipse.org/modeling/emf.

[2] Qvt specification : http ://www.omg.org/docs/formal/08-04-03.pdf.

[3] Semantics of business vocabulary and business rules (sbvr) 1.0 specification : http ://www.omg.org/spec/sbvr/1.0/.

[4] Philippe Blache and Jean-Marie Balfourier. Property grammars : a flexible constraint-based approach to parsing. In *IWPT*. Tsinghua University Press, 2001.

[5] Jordi Cabot, Robert Clarisó, and Daniel Riera. Umltocsp : a tool for the formal verification of uml/ocl models using constraint programming. In R. E. Kurt Stirewalt, Alexander Egyed, and Bernd Fischer, editors, *ASE*, pages 547–548. ACM, 2007.

[6] Trung T. Dinh-Trong, Sudipto Ghosh, and Robert B. France. A systematic approach to generate inputs to test uml design models. In *ISSRE*, pages 95–104. IEEE Computer Society, 2006.

[7] Alexander Felfernig, Gerhard Friedrich, Dietmar Jannach, and Markus Zanker. Configuration knowledge representation using uml/ocl. In Jean-Marc Jézéquel, Heinrich Hußmann, and Stephen Cook, editors, *UML*, volume 2460 of *Lecture Notes in Computer Science*, pages 49–62. Springer, 2002.

[8] Laurent Henocque and Mathieu Estratat. Parsing languages with a configurator. In *Proceedings of the European Conference for Artificial Intelligence ECAI'2004*, pages 591–595, Valencia, Spain, August 2004.

[9] Daniel Jackson. Automating first-order relational logic. In *SIGSOFT FSE*, pages 130–139, 2000.

[10] Ruth Raventós Jordi Cabot, Raquel Pau. From uml/ocl to sbvr specifications : a challenging transformation. *Information Systems Elsevier Journal*, 2009.

[11] Frédéric Jouault and Jean Bézivin. Km3 : A dsl for metamodel specification. In Roberto Gorrieri and Heike Wehrheim, editors, *FMOODS*, volume 4037 of *Lecture Notes in Computer Science*, pages 171–185. Springer, 2006.

[12] Frédéric Jouault and Ivan Kurtev. Transforming models with atl. In Jean-Michel Bruel, editor, *MoDELS Satellite Events*, volume 3844 of *Lecture Notes in Computer Science*, pages 128–138. Springer, 2005.

[13] Ulrich Junker. *Configuration*, volume Handbook of Constraint Programming, chapter 26. 2006.

[14] Ulrich Junker and Daniel Mailharro. The logic of (j)configurator : Combining constraint programming with a description logic. In *proceedings of IJCAI'03*, Acapulco, Mexico, 2003. Springer.

[15] Kaarel Kaljurand. Ace view - an ontology and rule editor based on controlled english. In Christian Bizer and Anupam Joshi, editors, *International Semantic Web Conference (Posters & Demos)*, volume 401 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2008.

[16] R. I. Kittredge. *Sublanguages and controlled languages*. Oxford University Press, 2003.

[17] Daniel Mailharro. A classification and constraint-based framework for configuration. *AI in Engineering, Design and Manufacturing, (12)*, pages 383–397, 1998.

[18] Sanjay Mittal and Brian Falkenhainer. Dynamic constraint satisfaction problems. In *Proceedings of AAAI-90*, pages 25–32, Boston, MA, 1990.

[19] Daniel Sabin and Eugene C. Freuder. Composite constraint satisfaction. In *Artificial Intelligence and Manufacturing Research Planning Workshop*, pages 153–161, 1996.

[20] Rolf Schwitter and Norbert E. Fuchs. Attempto controlled english (ace) a seemingly informal bridgehead in formal territory (poster abstract). In *JICSLP*, page 536, 1996.

[21] Timo Soininen, Ilkka Niemela, Juha Tiihonen, and Reijo Sulonen. Representing configuration knowledge with weight constraint rules. In *Proceedings of the AAAI Spring Symp. on Answer Set Programming : Towards Efficient and Scalable Knowledge*, pages 195–201, March 2001.

[22] Markus Stumptner. An overview of knowledge-based configuration. *AI Communications*, 10(2) :111–125, June 1997.

[23] Markus Stumptner and Alois Haselböck. A generative constraint formalism for configuration problems. In P. Torasso, editor, *Advances in Artificial Intelligence : Proceedings of the Third Congress of the Italian Association for Artificial Intelligence AI*IA '93*, pages 302–313. Springer, Berlin, Heidelberg, 1993.

Génération et contrôle autonomes d'opérateurs pour les algorithmes évolutionnaires

Jorge Maturana Frédéric Lardeux Frédéric Saubion

LERIA, Université d'Angers
{nom}@info.univ-angers.fr

Résumé

Les algorithmes évolutionnaires ont largement démontré leur utilité pour la résolution de problèmes combinatoires. Toutefois, leur utilisation pratique suppose de régler, d'une part, un certain nombre de paramètres fonctionnels et, d'autre part, de définir judicieusement les opérateurs qui seront utilisés pour la résolution. En effet, comme pour la majorité des méthodes métaheuristiques, les performances d'un algorithme évolutionnaire sont intrinsèquement liées à sa capacité à correctement gérer l'équilibre entre l'exploitation et l'exploration de l'espace de recherche. Récemment, de nouvelles approches ont vu le jour pour rendre ces algorithmes plus autonomes, notamment en automatisant le réglage et/ou le contrôle de paramètres. Nous proposons ici une nouvelle méthode dont l'objectif est double : d'une part nous souhaitons contrôler dynamiquement le comportement des opérateurs au sein d'un algorithme génétique, à travers leurs probabilités d'application et, d'autre part, nous souhaitons gérer un ensemble important d'opérateurs potentiels, dont l'utilisateur ne connaît pas a priori les performances, de manière également automatisée. Grâce à un mécanisme d'évaluation de l'état de la recherche en cours et de récompenses et de pénalités, le système devra identifier les opérateurs efficaces au détriment de ceux qui le sont moins. Nous expérimentons cette approche sur le problème SAT afin de démontrer qu'un algorithme autonome peut obtenir des performances similaires à celles d'un algorithme dédié, disposant d'opérateurs spécifiquement sélectionnés. Cette démarche vise finalement à libérer l'utilisateur de tâches fastidieuses de réglage et de l'expertise nécessaire à la conception d'algorithmes, souvent ad-hoc.

Abstract

Evolutionary algorithms have been efficiently used for solving combinatorial problems. However, their practical use induces to fix a set of functional parameters and also to define carefully the suitable operators for the resolution. As with the majority of metaheuristics methods, the performance of an evolutionary algorithm is intrinsically linked to its ability to properly manage the balance between exploitation and exploration of search space. Recently, new approaches have emerged to make these algorithms more independent, especially by automating the setting and / or control of parameters. We propose a new approach whose objective is twofold : on one hand we want to dynamically control the behavior of operators in a genetic algorithm, thanks to their probabilities of application and, on the other hand, we want to manage an important set of potential operators, whose performances are a priori unknown. Using a mechanism for evaluating the current state of search and a system of rewards and penalties, we identify the efficient operators and the bad ones. We test this methodology on the SAT problem to demonstrate that an algorithm can autonomously performs similar to a dedicated algorithm, whose operators have been specifically designed. This approach actuially aims to free the user from tedious setup tasks and from the often ad-hoc expertise, which is needed for the design of such solving algorithms.

1 Introduction

Les algorithmes évolutionnaires (AE) ont largement été utilisés pour l'optimisation discrète et continue, balayant un large spectre d'applications. Basés initialement sur les principes de l'évolution naturelle, les AEs permettent de gérer un ensemble de configurations d'un problème, modifiées progressivement au moyen d'opérateurs de variation et ce, afin de converger progressivement vers une solution optimale ou sous-optimale de bonne qualité. La métaphore évolutionniste amène à considérer ces configurations comme des individus formant une population qui évolue au moyen d'opérateurs génétiques, de mutation ou de croisement, selon leurs spécificités. Ce cadre général de résolution de problèmes s'inscrit dans le contexte des méthodes dites métaheuristiques et les principales

difficultés liés à la mise en oeuvre pratique de tels algorithmes reposent sur le choix d'un codage adéquat du problème ainsi que sur la définition d'opérateurs efficaces. Un fois cette architecture définie, le comportement de l'algorithme est en général déterminé par un ensemble de paramètres qui permettent d'agir principalement sur ses capacités à correctement explorer et exploiter l'espace de recherche, c'est à dire l'ensemble des configurations possibles du problème. Nous choisissons ici de distinguer les paramètres structurels de l'AE, tels que la taille de la population, des paramètres comportementaux, tels que les probabilités d'application des différents opérateurs. Le réglage de ces paramètres [11] s'avère alors être une tâche particulièrement ardue qui repose sur le savoir-faire, bien souvent empirique, de l'utilisateur et sur ses connaissances des caractéristiques du problème à résoudre. Dès lors, l'ajustement de paramètres s'appuie sur une série, en général coûteuse, d'expériences. Une telle approche réduit considérablement l'universalité des valeurs obtenues et la transposition de ces expérimentations à d'autres problèmes. Une autre voie consiste alors à envisager un contrôle des paramètres durant l'exécution de l'AE. Ce contrôle peut être supervisé par un ensemble de connaissances acquises au préalable (par exemple, choix d'une méthode de résolution dans les algorithmes de type *portfolio*) ou encore être abordé dans une optique plus autonome, les paramètres évoluant en fonction de l'état courant de la recherche. Ce nouveau paradigme visant à produire des algorithmes de résolution autonomes est en pleine émergence, à l'intersection de plusieurs domaines de l'informatique, notamment en intégrant des outils issus de l'apprentissage automatique, de l'optimisation combinatoire ou encore de la programmation par contraintes [1, 2, 9]. Mais, outre le contrôle des paramètres, le choix même des composants structurels de l'AE nécessite également une expertise de la part de l'utilisateur car, si des opérateurs de variations standard existent dans la littérature (comme le croisement uniforme par exemple), l'obtention de résultats acceptables est inévitablement conditionné par la spécialisation du schéma algorithmique général et la définition d'opérateurs appropriés.

Dans ce contexte, notre objectif est de proposer une nouvelle approche pour rendre les AE plus autonomes, à la fois dans le choix des opérateurs qu'ils utilisent mais également dans le réglage des paramètres qui leur sont intrinsèquement liés. Se basant sur des travaux précédents permettant de contrôler dynamiquement les probabilités d'application des opérateurs d'un AE [14, 12], notre approche peut être résumée par la figure 1.

Les performances de l'AE doivent être évaluées en fonction de mesures permettant de rendre compte de

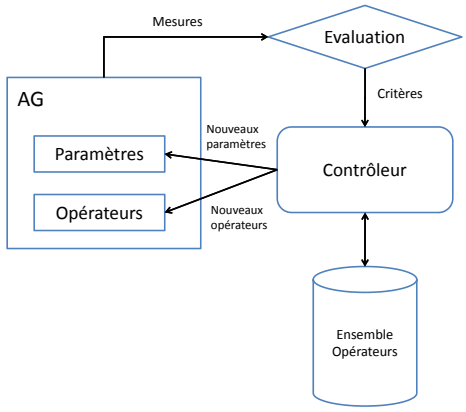


FIG. 1 – Schéma général

l'état de la recherche en cours. Deux notions sont traditionnellement mises en avant comme les clés du succès de ces techniques de recherche heuristiques de solutions dans un espace potentiellement très vaste : la diversification et l'intensification. Alors que la diversification rend compte de l'aptitude de la méthode à explorer des zones variées de l'espace de recherche, l'intensification correspond à la propension qu'a l'algorithme à converger vers une solution dans une zone précise. Dans le cadre des AE, nous mettons en avant deux mesures que nous avons précédemment utilisées pour contrôler cet équilibre entre intensification et diversification [13] : la qualité moyenne de la population et sa diversité (ici mésurée comme l'entropie des valeurs de vérité affectées aux variables). Ces deux mesures seront alors utilisées par un processus d'évaluation afin que le contrôleur puisse, d'une part, déterminer les nouvelles valeurs des paramètres et, d'autre part, choisir les opérateurs à conserver dans l'AE.

Dans cette nouvelle conception de l'AE, nous disposons d'un ensemble initial (qui pourrait également être généré dynamiquement par une "fabrique d'opérateurs") d'opérateurs qui peuvent être inclus dans l'AE à tout moment, leur taux d'application étant fixé par un paramètre idoine. Par conséquent, le principe général du contrôle consiste à distinguer les "bons" opérateurs, pour les appliquer davantage et favoriser leur action bénéfique, des "mauvais" pour les remplacer au sein de l'AE par de nouveaux candidats. La complexité de cette approche réside de manière duale dans l'évaluation de la situation en cours et dans la manière de récompenser ou de pénaliser les opérateurs.

Dans cet article, nous appuierons notre validation expérimentale sur la résolution du problème du problème canonique de satisfaction en logique propositionnelle (SAT) [7, 17]. Par le passé, nous avons pro-

posé un AE dédié au problème SAT (GASAT [10]) qui s'est avéré performant lors de la campagne d'évaluation SAT 2004. Forts de cette expérience, nous définissons ici un ensemble suffisamment large d'opérateurs de croisements (plus de 300) susceptibles d'être utilisés au sein d'un même AE, intégrant un mécanisme de contrôle autonome des probabilités d'applications de ces croisements. Notre objectif est double :

- Montrer que l'AE peut sélectionner de manière autonome des opérateurs adaptés à l'état courant de la recherche grâce au mécanisme de contrôle que nous lui adjoignons, et,
- Montrer que, sans connaissance préalable des opérateurs les plus performants, nous pouvons réobtenir, grâce au mécanisme de contrôle, des résultats comparables à ceux que nous avions obtenus avec notre algorithme GASAT, qui bénéficiait d'une analyse approfondie des meilleurs opérateurs possibles.

2 Contrôle autonome de paramètres pour les AEs

De nombreux travaux ont abordé le problème du réglage de paramètres pour les algorithmes évolutionnaires et nous renvoyons le lecteur à [11] pour un panorama plus exhaustif. Dans ce chapitre, nous nous concentrerons sur les éléments directement liés à la méthode que nous développons ici.

2.1 Réglage de paramètres pour les AEs

Le réglage de paramètre peut être abordé en se référant à la taxonomie proposée par Eiben et al. [4].

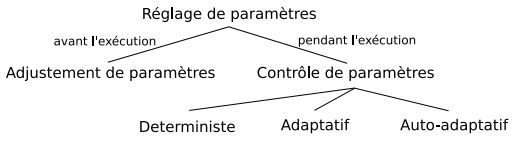


Fig. 2 – Taxonomie du contrôle de paramètre proposée par Eiben et al. [4]

Dans cette classification, on distingue les stratégies de réglage suivant qu'elles cherchent à fixer les valeurs des paramètres avant l'exécution de l'algorithme ou pendant celle-ci. Dans le cas d'un paramétrage initial, on cherche avant tout à trouver des valeurs universelles qui s'appliqueront sur la classe de problèmes la plus large possible. De tels résultats supposent de collecter un nombre important de mesures expérimentales et reposent en général sur une observation empirique ou, tout au moins, une extrapolation de phénomènes observés. S'agissant d'un réglage effectué pendant l'exécution même de l'algorithme, les auteurs distinguent ici trois branches selon le degré d'autonomie de la stratégie employée. Les approches déterministes reposent sur l'utilisation de règles prédéfinies alors que les approches adaptatives et auto-adaptatives cherchent à modifier les paramètres en fonction de l'observation de l'état courant du processus de recherche.

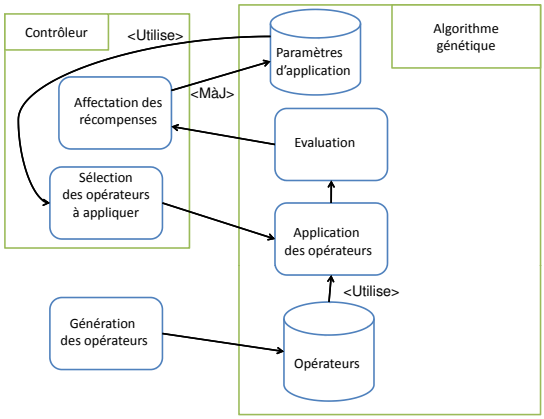


Fig. 3 – Approche générale pour la sélection adaptative d'opérateurs

Nous nous intéressons ici à la branche adaptative, qui, de manière plus ambitieuse, vise à libérer au maximum l'utilisateur du réglage et de la compréhension, plus ou moins intuitive, de l'impact des paramètres sur la recherche.

2.2 Sélection adaptative d'opérateurs (SAO)

Nous voulons décrire à présent les principes de bases qui ont émergé ces dernières années pour concevoir des algorithmes évolutionnaires capables de gérer leurs opérateurs de manières plus autonomes. Le schéma de la figure 3 permet de distinguer les deux phases essentielles, utilisées dans les approches qui nous intéressent ici : la récompense des opérateurs et la sélection de ces mêmes opérateurs relativement à la manière dont ils ont été récompensés..

La récompense, matérialisée par un crédit qui sera affecté aux opérateurs, s'appuie sur une évaluation de l'état courant de la recherche. La plupart des méthodes s'appuient souvent sur une utilisation exclusive de la qualité des populations (moyenne, meilleur individu...) [18, 5]. Dans des travaux précédents, nous avons proposé d'utiliser conjointement la qualité moyenne de la population et sa diversité [13]. Une fois les critères d'évaluation fixés, l'affectation de crédits aux opérateurs peut alors être plus où moins sophistiquée (utilisation de fenêtres de valeurs, calcul de valeurs moyennes ou utilisation de valeurs extrêmes). L'objec-

tif principal est de proposer le système de récompenses le plus générique et le plus robuste possible mais qui soit également suffisamment sensible aux changements d'états afin de permettre au mécanisme de sélection d'opérateur d'être réactif. Les crédits peuvent être vus comme un moyen d'apprendre dynamiquement des informations sur la qualités des opérateurs au cours de la résolution.

La sélection d'opérateurs va s'appuyer sur les récompenses obtenues précédemment pour indiquer à l'algorithme quel(s) opérateur(s) utiliser lors du prochain état basique de calcul, réglant ainsi les paramètres initiaux de l'AE. Ici encore, les choix sont multiples allant d'une probabilité d'application proportionnelle à la récompense à des modèles plus sophistiqués, issus de la théorie des jeux. La question principale est de proposer un système de contrôle qui permette de rester réactif et adaptatif, ne convergeant pas trop vers une utilisation systématique d'opérateurs au comportement "moyen".

3 Opérateurs de croisements pour SAT

3.1 Le problème SAT

Le problème SAT [7, 17] consiste à trouver une affectation satisfaisant une expression Booléenne. Une instance de ce problème est donc une formule Booléenne qui peut être mise sous forme normale conjonctive (CNF), c'est à dire sous la forme d'une conjonction de clauses où les clauses sont des disjonctions de littéraux (variables ou négations de variable). Quand toutes les clauses peuvent être satisfaites, le problème est dit satisfiable. Dans le cas contraire, il est souvent intéressant de minimiser le nombre de clauses fausses. Deux familles de méthodes permettent de résoudre ces problèmes : les méthodes exactes qui répondent au problème de décision (satisfiable ou non satisfiable) et les méthodes approchées qui répondent au problème d'optimisation (minimiser le nombre de clauses fausses).

3.2 Algorithmes évolutionnaires pour SAT

Les AEs pour SAT [3, 6, 8, 16, 10] font partie des méthodes approchées. Comme cela a été rappelé dans l'introduction, le principe de base de ces algorithmes est de manipuler une population d'individus à l'aide d'opérateurs génétiques afin d'obtenir des individus de bonne qualité. Dans le cas du problème SAT, les individus sont des affectations des variables Booléennes (i.e., d'interprétations logiques possibles) et l'objectif est d'avoir des individus induisant le moins de clauses fausses possibles. Plusieurs types d'opérateurs sont utilisés au sein des AEs : sélection, croisement, mutation, insertion ...

L'algorithme génétique pour SAT que nous allons utiliser pour nos expériences repose sur GASAT [10] qui est actuellement l'un des AEs les plus efficaces pour SAT. Son principe basique est le suivant :

1. une population est aléatoirement générée ;
2. deux individus sont sélectionnés aléatoirement dans une sous population des 15 meilleurs individus ;
3. l'opérateur de croisement CC (défini section 3.3) est appliqué sur ces individus ;
4. l'opérateur de mutation applique une recherche locale sur le fils obtenu ;
5. le nouvel individu est inséré dans la population et remplace le plus vieil individu s'il est meilleur que le moins bon individu de la sous population ;
6. si aucune des conditions d'arrêt (affectation satisfaisant le problème, nombre de croisements maximum atteint, ...) n'est atteinte, retour à 2.

L'objectif de notre travail étant de mettre en avant les propriétés du contrôleur, nous avons modifié GASAT pour n'en garder que le squelette et ainsi observer la seule action du contrôleur. Pour cela, nous avons remplacé l'opérateur de sélection par une sélection aléatoire et nous avons supprimé la mutation. L'opérateur d'insertion est quant à lui modifié pour qu'il substitue automatiquement, au plus vieil individu, le fils nouvellement créé.

3.3 Une famille d'opérateurs de croisement

L'opérateur de croisement est un opérateur très important. Il permet à la population d'être régénérée avec de bons individus. Dans notre problématique de contrôle autonome, la définition d'un bon individu dépend de l'état de la recherche. Dans certains cas, il peut être plus intéressant pour la recherche d'obtenir un individu qui diversifie la population et dans d'autres, les individus améliorant la qualité seront les plus attendus. Pour cette raison, nous avons décidé de travailler avec plusieurs croisements ayant chacun une prédisposition, soit à améliorer la qualité, soit à favoriser la diversité. La majorité des croisements existants essaie de conserver les bonnes propriétés des parents afin de les reproduire chez le fils. Par exemple :

- le croisement uniforme conserve les valeurs de vérité des variables qui sont identiques chez les deux parents ;
- le croisement proposé par Fleurent et Ferland [6] utilise l'ensemble des clauses qui sont vraies chez un parent et fausses chez l'autre. Seules sont conservées les valeurs des variables apparaissant dans les clauses vraies de cet ensemble ;

– le croisement CC [10] ne traite que les clauses fausses simultanément chez les deux parents et les rend vraies chez le fils en flippant une variable dans chacune d'elles ;
– le croisement CCTM [10] opère de la même manière que CC mais il travaille ensuite sur les clauses vraies chez les deux parents afin de garantir que les clauses seront aussi vraies chez le fils.

Très peu d'opérateurs de croisements ont comme objectif principal la diversification. Nous en avons donc redéfini plusieurs qui essaient de détecter les bonnes propriétés des parents pour ensuite les casser. Par exemple :
– le croisement uniforme "inverse" qui flippe toutes les valeurs de vérité des variables qui sont identiques chez les deux parents ;
– le croisement NT ne traite que les clauses vraies chez les deux parents et les rend fausses chez le fils.

Nous avons proposé et utilisons beaucoup d'autres croisements (307 au total) qui sont des combinaisons de croisements existants. Dans cet ensemble se trouve des croisements dont l'objectif principal est de diversifier, d'autres dont la tâche première est d'améliorer la qualité et enfin un grand nombre dont l'action est moins tranchée.

4 Description du contrôleur

Dans cette section nous allons décrire précisément le mécanisme de contrôle de l'AE que nous proposons. L'architecture globale de notre approche est détaillée par la figure 4, sur laquelle on constate que le contrôleur comprend principalement deux composants :
– la sélection adaptative d'opérateur (SAO) qui, comme présentée dans la section 2, constitue l'interface de commande permettant de déterminer quel est l'opérateur à appliquer (*Sélection d'opérateurs* (SO)) et de collecter la rétroaction de l'AE, c'est à dire l'évaluation de l'impact de l'opérateur sur la recherche en cours afin de lui attribuer une récompense (*Affectation de crédits* (AC)).
– Le *forgeron* qui est responsable de fournir des opérateurs adéquats pour l'AE. Ce forgeron se base sur une spécification générale de schémas d'opérateur et sur les diverses options possibles que ces opérateurs peuvent combiner.

Bien que l'on puisse considérer que ces composants contrôlent tous deux des paramètres de l'AE, il existe pourtant une différence fondamentale entre leurs actions respectives. Tandis que la SAO contrôle des paramètres comportementaux (i.e., choisir la probabilité d'application d'un opérateur à un moment donné), le forgeron modifie des paramètres structurels (choisir quels opérateurs seront insérés). Les choix de la SAO sont donc dépendants de ceux du forgeron.

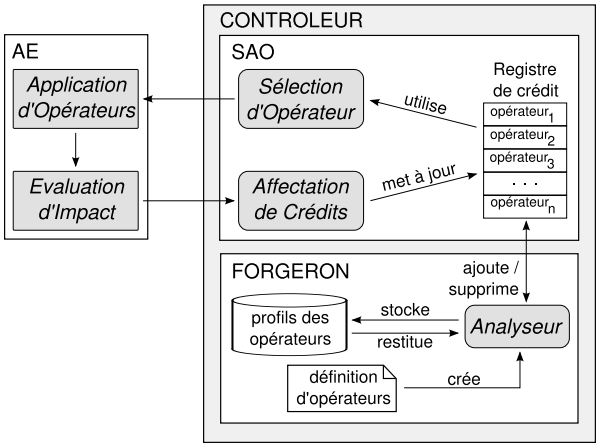


FIG. 4 – Schéma général du contrôleur

4.1 Affectation de crédits

Comme mentionné dans la section 2, et afin d'évaluer la performance des opérateurs, il faut mesurer leurs performances après application. La méthode présentée dans [12, 14] prend en compte trois mesures différentes : la variation de diversité, la variation de qualité et le temps d'exécution. L'objectif est alors de maximiser les deux premiers critères et de minimiser le troisième. Comme ces objectifs sont contradictoires, nous avons besoin de trouver un façon de les combiner pour obtenir une évaluation unique.

La méthode *Compass (C)*, que nous avions proposé dans [14] (Figure 5.a), prend en considération la distance d'un point $(\Delta Diversité, \Delta Qualité)$, représentant l'évaluation d'un opérateur o, à une ligne incliné d'un angle $\Theta = \pi/4$. Le point le plus à droite et en haut est le meilleur. Puis, ces valeur comparatives sont divisées par le temps d'exécution pour récompenser les opérateurs les plus rapides.

Dans cet article, deux autres principes de mesure ont été comparés, tous deux basés sur la notion de dominance Pareto [15]. De manière succincte, lorsque deux points sont comparés suivant plusieurs critères ($\Delta Diversité$ et $\Delta Qualité$ dans notre cas), on dit qu'un point *domine* l'autre s'il est meilleur (plus grand dans notre cas) sur tous les aspects. Dans le cas contraire, les points sont incomparables.

L'affectation de crédits dénommée *Dominance Pareto (DP)* considère le nombre d'opérateurs que chaque opérateur domine (Figure 5.b) ; par contre, pour la mesure *Rang Pareto (RP)*, on regarde combien d'opérateurs dominent chaque opérateur (les va-

leurs les plus petites sont alors préférables). Une différence importante existe entre ces deux évaluations. Alors que le RP considère uniquement des opérateurs qui ne sont pas dominés, la DP récompense de ceux qui sont en forte concurrence avec d'autres opérateurs. Un effet de type essaim se produit ici : il ne suffit pas d'être bon mais aussi il faut être bon là où la plupart des opérateurs sont placés.

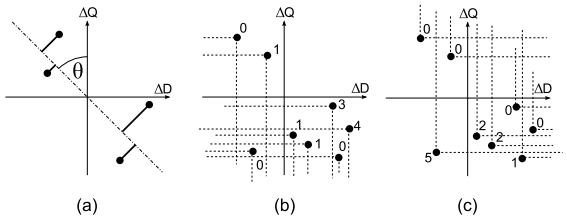


FIG. 5 – Schémas d'affectation de crédits. Compass (a), Dominance Pareto (b), Rang Pareto (c)

4.2 Sélection d'opérateurs

L'idée de la SO présentée dans [12], appelée Ex-DMAB, est inspirée des méthodes de bandits manchots (MAB) utilisées en théorie de jeux. Ici, une stratégie choisit toujours l'opérateur le plus performant, en même temps qu'elle assure de ne pas retarder indéfiniment l'invocation des autres opérateurs. Afin de s'adapter aux environnements dynamiques, comme c'est le cas pour les AEs, un test statistique surveille les changements des séries temporelles associées aux opérateurs, et réinitialise la mémoire du MAB lors d'un changement de comportement évident comme, par exemple, lorsque l'actuel meilleur opérateur est dépassé par un autre.

Dans le travail que nous présentons ici, nous comparons quatre SO différentes :

MAB2 , inspiré de Ex-MAB [5], mais adaptée pour bien fonctionner avec des ensembles dynamiques d'opérateurs. En effet, la méthode présentée dans [5] se base sur le nombre de fois où les opérateurs ont été utilisés pour n'"oublier" aucun d'entre eux. Ceci ne fonctionne pas bien lorsque de nouveaux opérateurs viennent d'être introduits dans l'AE, car il faudrait les appliquer un certain nombre de fois pour qu'ils rattrapent les autres. Nous proposons alors un mécanisme qui consiste à considérer le nombre de générations écoulées depuis la dernière application de chacun. De plus, pour des raisons de simplicité, le mécanisme de réinitialisation n'est pas utilisé.

Aléatoire (A) , choisit au hasard l'opérateur à appliquer.

MAB2+Détection de Collusion (M2C) , similaire à MAB2, il comporte en plus un mécanisme de détection de collusion des individus (nous choisissons ce terme, sans doute un peu fort, traduisant le regroupement autour des optima locaux, qui nuit à la poursuite de la recherche). Ceci est réalisé en considérant la pente de la ligne résultant de la régression linéaire des valeurs de qualité moyenne de la population. Si la pente est proche de zéro, on considère que l'AE est "coincé"' et on démarre une phase d'exploration forcée, en ne prenant que les opérateurs qui se sont avérés être exploratoires. Cette phase est abandonnée lorsque la diversité atteint une certaine fraction au-dessus de la diversité actuelle, quand il n'y a pas d'opérateurs de diversification, ou quand un nombre de générations sans pouvoir augmenter la diversité s'est écoulé.

Probabilité Proportionnelle (PP) choisit les opérateurs avec une probabilité proportionnelle aux récompenses enregistrées dans le registre de crédit.

4.3 Forgeron

Le Forgeron, comme évoqué en section 2, constitue la "fabrique d'opérateurs", qui contrôle alors les paramètres structurels correspondant aux opérateurs présents à chaque instant de l'exécution de l'AE. Il remplit les fonctions suivantes :

Crée (forge) des opérateurs, à partir de leur définition/spécification,

Ajoute les opérateurs au registre de crédit, en fournissant à la SAO de nouveaux choix d'opérateurs,

Analyse la performance des opérateurs dans le registre, pour éventuellement supprimer un opérateur de l'AE ou en ajouter un nouveau,

Supprime les opérateurs les moins performants en fonction de l'information stockée dans le registre de crédit,

Stocke le profil des opérateurs éliminés, afin d'avoir une trace de leur existence, pour éventuellement les ré-insérer dans le registre plus tard,

Restitue des opérateurs éliminés en cas de besoin.

Les opérateurs peuvent donc avoir trois états principaux : *non-nés*, avant qu'ils ne soient créés pour la première fois ; *vivants*, quand ils sont placés dans le registre de crédit et sont utilisés par la SAO, et *morts*, après avoir été éliminés du registre.

L'élimination d'un opérateur ne signifie nécessairement pas qu'il soit mauvais *per se*, mais plutôt dans le contexte des nécessités actuelles de la recherche : un opérateur peut être un excellent diversificateur, mais être éliminé si l'état actuel de la recherche a besoin d'opérateurs d'exploitation pour converger davantage.

Ceci explique pourquoi les profils des opérateurs sont stockés lorsqu'ils sont éliminés du registre : en cas de besoin, la résurrection d'un opérateur ne se fera pas aveuglément.

Dans notre implémentation, le registre de crédit est composé d'un nombre fixe d'opérateurs, et évalué à intervalles réguliers par le forgeron dans la recherche des opérateurs faibles qui puissent être éliminés pour laisser place à un nouveau né. Tous les opérateurs possibles sont créés puis essayés avant de rechercher à nouveau parmi les morts, ce qui permet de donner à tous une chance de démontrer leurs capacités.

5 Étude expérimentale

Nous présentons ici une analyse expérimentale de la méthode décrite plus haut, dans le cadre de la résolution du problème SAT.

5.1 Cadre d'application

Pour nos expériences nous avons sélectionné des instances venant de différentes compétitions SAT. Le choix des benchmarks essaie de couvrir les différentes familles d'instances (aléatoires, faites-main et industrielles) : *f500* est une instance générée aléatoirement au seuil ; *aim-100-1_6-yes1-1* (notée aim-100 par la suite) est une instance aléatoire modifiée afin de n'avoir qu'une seule solution ; *3bitadd_31.shuffled* (notée 3bitadd par la suite) est une instance de VLSI ; *ibm-2004-29-k55* (notée ibm par la suite) est une instance industrielle (BMC) ; *simon-s02b-r4b1k1.2* (notée simon par la suite) est une instance difficile des compétitions SAT.

L'algorithme de base servant à nos expériences (voir section 3.2) est appliqué 50 fois pour chaque croisement et chaque contrôleur et cela pour chacune des 5 instances testées. Lors d'une exécution, la population possède 100 individus et le nombre de croisements autorisé est de 10^5.

5.2 Réglage du contrôleur

L'objectif ici est d'essayer les différentes combinaisons de mécanismes d'affectation de crédits et de sélection d'opérateurs présentés en 4.1 et 4.2. Ces combinaisons seront identifiés par la notation $X - Y$, où $X \in \{C, DP, RP\}$ est le mécanisme d'AC, et $Y \in \{M2, A, M2C, PP\}$ est le mécanisme de SO (voir section 4).

Les paramètres du contrôleur sont ceux des SOs MAB2 et M2C ainsi que ceux du Forgeron. Du coté de MAB2, l'opérateur à appliquer sera celui qui maximise l'expression $MAB2_{o,t}$, correspondant à l'opérateur o à l'instant actuel t qui est donné par la formule suivante.

$$MAB2_{o,t} = r_{o,t} + 2 \cdot exp(p \cdot a_{o,t} - p \cdot x \cdot trc_t) \quad (1)$$

où $r_{o,t}$ est la récompense de l'opérateur o à l'instant t, correspondant à la valeur maximale des 10 dernières applications, $a_{o,t}$ est le temps d'*oubli* (i.e., le nombre de générations écoulées depuis la dernière application de o), trc_t correspond à la taille du registre de crédit à l'instant t. x est un paramètre qui indique combien de générations doit laisser passer un paramètre dans la phase d'oubli avant d'être forcement appliqué, exprimé comme un facteur de la taille du registre de crédit ($x > 1$). La force de cette exploration est exprimée par une exponentielle, qui n'a presque aucune importance au début, mais augmente rapidement à la fin de la période définie par x. p influence la croissance de cette exponentielle.

La formule 1 comporte deux parties. Celle de gauche encourage l'application des opérateurs qui ont reçu les meilleures récompenses, et celle du droite encourage les opérateurs qui n'ont pas été appliqués depuis longtemps. Les valeurs de $r_{o,t}$ sont normalisées dans l'intervalle $[0, 1]$. La partie de droite donne une valeur proche de zéro, sauf quand $a_{o,t}$ se rapproche de $x \times trc_t$, implicant, dans ce cas, une valeur de 2. En pratique, aucun opérateur n'est abandonné plus de $(x \times trc_t)$ générations. Les valeurs pour les paramètres lors de nos expériences étaient $trc_t = 20$, $x = 1.5$ et $p = 0.2$.

Dans MAB2+DC, on trouve de plus les paramètres de détection de collusion. La phase d'exploration est décrétée si dans les 100 dernières générations, la différence entre la plus grande et la plus petite des qualités moyennes de la population est inférieure à 0.001 et la pente de la ligne résultant de la régression linéaire est comprise dans ± 0.0001.

On pourrait objecter ici que nous remplaçons les paramètres initiaux d'application des opérateurs par de nouveaux paramètres tout aussi délicats à régler. Toutefois, nous avons clairement mesuré qu'un moins bon réglage du contrôleur avait un impact bien moindre sur la performance de l'algorithme qu'un mauvais réglage de ses paramètres comportementaux. D'autre part, nous travaillons ici avec quelques centaines d'opérateurs et donc quelque centaines de paramètres initiaux (sans même évoquer l'espace des algorithmes possibles induit par la combinaison de ces opérateurs), alors que nous devons fixer trois ou quatre paramètres pour le contrôleur.

En ce qui concerne le Forgeron, le registre de crédit a un taille fixe de 20 opérateurs. Toutes les 50 générations, le forgeron analyse le registre dans le but de trouver un opérateur faible pour le supprimer et le remplacer par un opérateur non-né ou mort. Si un opérateur a un nombre suffisant d'applications ($\frac{1}{2}$ de

la taille de la fenêtre, c'est à dire 5 applications) et que sa récompense est dans le tiers inférieur par rapport au reste, il est choisi pour être éliminé.

6 Résultats

Nous présentons ici le compte-rendu expérimental des résultats obtenus avec notre approche, en tentant d'en illustrer les aspects les plus frappants. La figure 6 montre la convergence moyenne sur 50 exécutions du meilleur individu pour les différents combinaisons d'AC et de SO pour l'instance *ibm*. L'abcisse correspond au générations écoulées (chaque génération correspondant à une application d'opérateur).

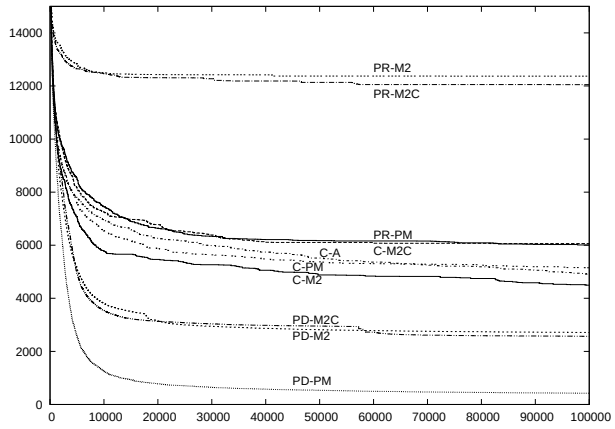


FIG. 6 – Nombre de clauses fausses du meilleur individu sur instance *ibm*, avec différents contrôleurs

Ici, les meilleurs résultats correspondent aux contrôleurs DP-PP, DP-M2 et DP-M2C. La figure 7 montre la diversité moyenne sur 50 exécutions pour ces contrôleurs sur la même instance. Notons que des contrôleurs qui obtiennent des niveaux similaires en terme de qualité, ne maintiennent pas forcement les mêmes niveaux de diversité. DP-PP se caractérise par son instabilité en terme de diversité, dûe à l'utilisation de la SO PP, qui, par rapport à M2 et M2C, induit une capacité d'exploration supérieure. Ceci explique la bonne performance de DP-PP sur toutes les instances, comme on le verra par la suite.

La tendance à la hausse de la diversité pour DP-PP à partir de la génération 8000 est dûe à la prise en compte de la diversité dans l'évaluation des opérateurs. Au début de la recherche, la population initiale est composée d'individus générés aléatoirement. Ceci facilite la tâche des opérateurs "exploitatoires", qui font décroître la diversité en même temps que la qualité augmente. Cependant, à partir d'un certain moment, l'amélioration devient difficile, et le contrôleur se tourne vers l'exploration, en cherchant à accroître la

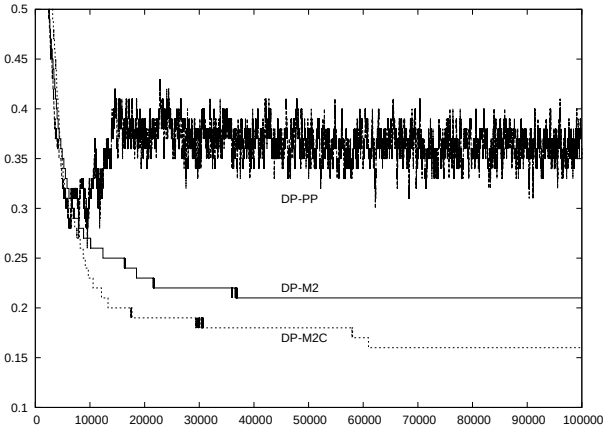


FIG. 7 – Diversité des trois meilleurs contrôleurs sur instance *ibm*

diversité afin que les individus puissent s'échapper des optima locaux. Ce comportement est précisément celui que l'on cherche en incluant la diversité dans l'évaluation qui est transmise au contrôleur.

Cette tendance est particulièrement forte dans l'AC C. La figure 8 montre l'évolution de la diversité moyenne sur 50 exécutions pour les différents méthodes d'AC : C, DP et RP utilisées conjointement avec la SO M2C, sur l'instance *simon*. Dans cette figure on peut apprécier la, sans doute, excessive exploration induite par C, ce qui expliquerait que ces résultats ne soient pas les meilleurs de la série.

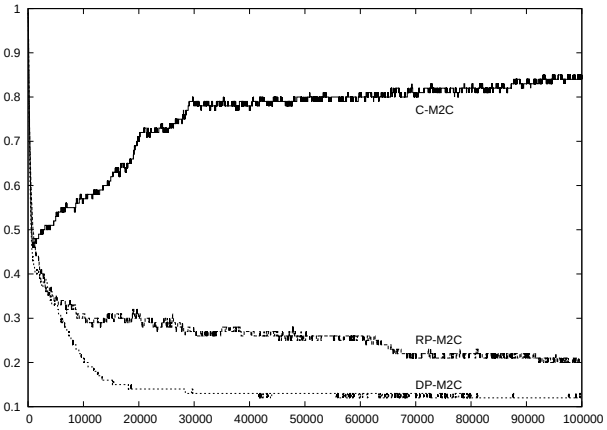


FIG. 8 – Diversité des CAs C, DP et RP avec SO M2C, montrant le passage de l'exploitation vers l'exploration avec différentes méthodes d'AC

La table 6 indique les moyennes des résultats ainsi que l'écart type (entre parenthèses) sur 50 exécutions des différents contrôleurs, ainsi que les exécutions avec les opérateurs Uniforme, FF, CC et CCTM. Les meilleurs résultats sont marqués en gras.

	f500		aim-100		3bitadd		ibm		simon	
Unif	218.4	(5.7)	11.2	(1.0)	5781.0	(141.5)	34149.6	(138.5)	2872.6	(33.9)
FF	30.2	(4.9)	1.9	(0.6)	164.7	(24.5)	3827.8	(160.5)	137.5	(9.7)
CC	7.2	(1.3)	1.9	(0.6)	12.2	(3.2)	1247.7	(98.67)	81.6	(5.4)
CCTM	7.3	(1.4)	1.8	(0.6)	**12.2**	**(2.6)**	1237.2	(78.1)	**81.2**	**(5.3)**
C-M2	25.9	(24.0)	2.4	(1.9)	469.6	(83.7)	6009.7	(3024.6)	189.0	(17.3)
C-M2C	16.8	(19.7)	2.6	(1.9)	519.7	(80.0)	6063.4	(2171.8)	194.4	(23.9)
C-PP	56.3	(33.6)	2.2	(1.9)	481.9	(137.5)	5151.9	(2758.8)	209.2	(41.8)
DP-M2	67.4	(62.7)	3.3	(2.2)	416.5	(453.7)	2712.0	(3523.9)	94.3	(103.0)
DP-M2C	53.3	(59.0)	2.5	(1.5)	266.0	(405.4)	2567.1	(4206.3)	108.0	(102.5)
DP-PP	**6.0**	**(1.4)**	**1.0**	**(0.0)**	303.3	(47.7)	**423.8**	**(75.2)**	93.5	(7.7)
RP-M2	98.2	(53.8)	2.9	(1.7)	534.6	(478.9)	12370.3	(5214.2)	201.0	(188.7)
RP-M2C	104.2	(52.5)	2.6	(1.8)	395.2	(431.3)	12050.3	(5141.1)	277.9	(200.0)
RP-PP	7.8	(1.5)	1.0	(0.0)	517.1	(51.8)	4495.9	(791.9)	148.9	(11.9)
C-A	21.0	(14.4)	1.1	(0.2)	404.9	(59.7)	4908.4	(1623.0)	183.7	(16.7)

TAB. 1 – Nombre de clauses fausses et écart type

Le meilleurs résultats obtenus pour un contrôleur sont ceux de DP-PP, qui améliorent le meilleur des croisements de l'état de l'art dans trois instances. La table 6 montre les pourcentages d'amélioration de DP-PP.

	f500	aim-100	3bitadd	ibm	simon
Unif	**97,23%**	**91,07%**	**94,75%**	**98,76%**	**96,75%**
FF	**80,01%**	**47,92%**	-84,13%	**88,93%**	**31,99%**
CC	**16,57%**	**47,92%**	-2381,67%	**66,03%**	-14,62%
CCTM	**17,03%**	**45,05%**	-2385,74%	**65,74%**	-15,15%

TAB. 2 – Amélioration du contrôleur DP-PP par rapport aux croisements de l'état de l'art.

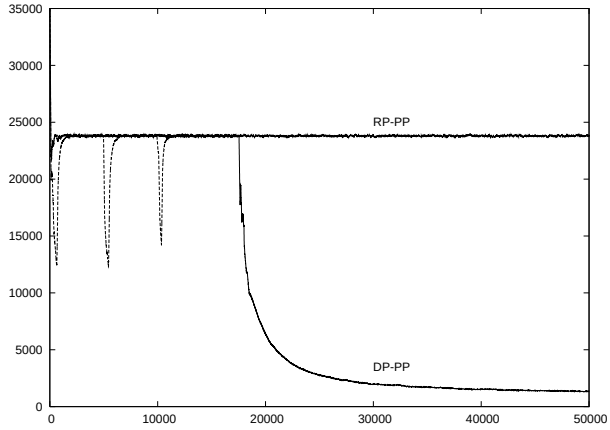


FIG. 9 – Comportement de Rang Pareto versus Dominance Pareto : Nombre moyenne de clauses fausses sur l'instance *ibm*

On peut remarquer que le contrôleur DP-PP produit des résultats comparables à ceux obtenus avec les meilleurs opérateurs sans contrôleur (sauf pour *3bitadd*). Toutefois, rappelons que la mise au point de CC et CCTM (les meilleurs opérateurs sans contrôleur) repose sur un travail de plusieurs semaines de comparaisons, d'analyses et d'expériences [10], tandis que le contrôleur réalise un travail similaire, en quelques minutes et, surtout, sans intervention humaine. De plus, si on considère Notamment l'instance industrielle *ibm*, le nombre de clauses fausses moyenne trouvées avec DP-PP équivaut à $\frac{1}{3}$ de ceux atteints avec CC et CCTM.

En comparant les différentes méthodes d'AC, on peut noter que, en général, DP apparaît dans la plupart des cas comme le contrôleur le plus performant, suivi par C puis RP. On pourrait se demander d'abord quelle est la différence entre les deux contrôleurs fondés sur les notions de la dominance Pareto. Pour répondre à cela il faut étudier leur comportement lors des exécutions individuelles. La figure 9 montre la qualité moyenne de la population en utilisant DP-M2 et RP-M2 sur l'instance *ibm*.

RP considère également tous les opérateurs placés dans la frontière de Pareto (points dans la figure 5.c avec une valeur égale à 0), ce qui induit un équilibre entre les tendances à explorer et à exploiter et empêche l'AE de pencher d'un coté plutôt que de l'autre. Notons que les tentatives d'augmentation de la qualité de RP-PP sont modérées, en provocant le retour de la qualité moyenne vers une valeur intermédiaire. Par contre, si on utilise DP, le fait de mieux considérer les opérateurs qui sont dans la tendance générale (points dans la figure 5.b avec une grande valeur) permet l'abandon de ce *statu quo* et d'améliorer finalement la qualité de la population. Cet "équilibre flexible" est le principal atout de cette méthode d'AC.

En ce qui concerne les méthodes de SO, on trouve un avantage marqué pour PP. M2 semble être également avantageux sur *simon*, mais en général son application ne produit pas de résultats concluants, ce qui est également le cas pour M2C. D'une façon étonnante, A n'est pas décevant (il faut se rappeler que cette SO choisit parmi les opérateurs que le Forgeron autorise à survivre).

Étant donné que PP et A produisent tous deux de bons résultats, on peut en déduire que l'exploration de

M2 et M2C a été trop conservatrice, et qu'on pourrait tirer plus de profits avec un réglage qui ne soit pas aussi focalisé sur l'exploitation. En général, on peut conclure que les aspects les plus importants du contrôleur sont le contrôle des paramètres structuraux (i.e., le choix des opérateurs disponibles pendant la recherche), l'évaluation de ces opérateurs (i.e., la méthode de CA) et finalement leur application (mécanisme de la SO).

7 Conclusion

Dans cet article nous avons proposé un contrôleur permettant de créer et d'appliquer de manière autonome des opérateurs de croisements pour les AEs. Ce contrôleur s'appuie sur la qualité des individus de la population mais prend aussi en compte le contrôle de la diversité. De façon plus générale, tout type de paramètres associés aux probabilités d'application des opérateurs peut être intégrés dans ce contrôleur.

Afin d'évaluer l'apport d'un tel mécanisme, nous l'avons intégré dans un AE travaillant sur le problème SAT. Le contrôleur a été comparé aux meilleurs opérateurs de croisement de l'état de l'art et il s'est avéré très compétitif voire même meilleur dans certains cas. De plus, grâce à l'automatisation du processus de conception, aucune expérience n'a été nécessaire pour choisir les opérateurs à fournir au contrôleur.

Des pistes pour un travail futur sont l'élaboration de schémas plus raffinés pour le contrôle des paramètres structuraux et l'incorporation d'un mécanisme de contrôle de la recherche [13]. Il serait aussi intéressant d'analyser les opérateurs les plus utilisés par DP-PP afin de vérifier l'intérêt du contrôleur en tant qu'outil d'assistance à la conception. Finalement, on peut envisager l'application de ce contrôleur sur d'autres opérateurs et algorithmes, tels que le Tabu Search ou le Simulated annealing).

Références

[1] R. Battiti and M. Brunato. *Learning and Intelligent Optimization. Proc. Int. Conf., LION 2007*, volume 5313 of *LNCS*. Springer, 2008.

[2] R. Battiti, M. Brunato, and F. Mascia. *Reactive Search and Intelligent Optimization*, volume 45 of *Operations Research/Computer Science Interfaces*. Springer Verlag, 2008.

[3] K.A. De Jong and W.M. Spears. Using GA to solve NP-complete problems. In *Proc. of Int. Conf. on GA ICGA*, pages 124–132, 1989.

[4] A. E. Eiben, R. Hinterding, and Z. Michalewicz. Parameter control in evolutionary algorithms. *IEEE Trans.Evol.Computation*, 3 :124–141, 1999.

[5] A. Fialho, L. Da Costa, M. Schoenauer, and M. Sebag. Extreme value based adaptive operator selection. In G. Rudolph et al., editor, *Proc. PPSN'08*, pages 175–184. Springer, 2008.

[6] C. Fleurent and J.A. Ferland. Object-oriented implementation of heuristic search methods for graph coloring, maximum clique, and satisfiability. In *Cliques, Coloring, and Satisfiability*.

[7] M.R. Garey and D.S. Johnson. *Computers and Intractability , A Guide to the Theory of NP-Completeness*. W.H. Freeman & Company, San Francisco, 1979.

[8] J. Gottlieb and N. Voss. Adaptive fitness functions for the SAT problem. In *Proc. PPSN*, pages 621–630. Springer Verlag, 2000. LNCS 1917.

[9] Y. Hamadi, E. Monfroy, and F. Saubion. Special issue on autonomous search. *Constraint Programming Letters*, 4, 2008.

[10] F. Lardeux, F. Saubion, and J-K Hao. GASAT : A genetic local search algorithm for the sat problem. *Evolutionary Computation*, 14(2) :223–253, 2006.

[11] F. Lobo, C. Lima, and Z. Michalewicz, editors. *Parameter Setting in Evolutionary Algorithms*, volume 54 of *Studies in Computational Intelligence*. Springer, 2007.

[12] J. Maturana, A. Fialho, F. Saubion, M. Schoenauer, and M. Sebag. Compass and dynamic multi-armed bandits for adaptive operator selection. In *Proc. of IEEE Congress on Evolutionary Computation CEC*, 2009. to appear.

[13] J. Maturana and F. Saubion. On the design of adaptive control strategies for evolutionary algorithms. In *Proc. Int. Conf. on Artificial Evolution. LNCS 4926, Springer*, 2007.

[14] J. Maturana and F. Saubion. A compass to guide genetic algorithms. In G. Rudolph et al., editor, *Proc. PPSN'08*, pages 256–265. Springer, 2008.

[15] V. Pareto. Cours d'économie politique. in Vilfredo Pareto, Oeuvres complètes, Genève : Librairie Droz, 1896.

[16] C. Rossi, E. Marchiori, and J.N. Kok. An adaptive evolutionary algorithm for the SAT problem. In *Proc. of Symposium on Applied Computing SAC*, pages 463–470. ACM press, 2000.

[17] L. Sais. *Problème SAT : progrès et défis*. Collection Program. par contraintes. Hermès, 2008.

[18] D. Thierens. *Parameter Setting in Evolutionary Algorithms*, chapter Adaptive Strategies for Operator Allocation, pages 77–90. Volume 54 of Lobo et al. [11], 2007.

Un modèle réactif pour l'optimisation par colonies de fourmis : application à la satisfaction de contraintes

Madjid Khichane[1,2], **Patrick Albert**[1] **et Christine Solnon**[2]

[1] ILOG An IBM Company
[2] Université de Lyon
Université Lyon 1, LIRIS CNRS UMR5205, France
{mkhichane,palbert}@fr.ibm.com,christine.solnon@liris.cnrs.fr

Résumé

Les algorithmes basés sur l'optimisation par colonies de fourmis (Ant Colony Optimization / ACO) sont paramétrés par deux coefficients α et β qui déterminent respectivement l'importance de l'expérience passée et de l'heuristique dans la probabilité de transition utilisée pour construire des solutions. Nous introduisons une méthode pour adapter dynamiquement les valeurs de ces deux paramètres qui ont une influence déterminante sur l'intensification et la diversification de la recherche. Cette méthode est elle-même basée sur ACO : des traces de phéromone sont associées aux valeurs potentielles des paramètres ; ces traces représentent l'expérience passée concernant l'utilisation de ces valeurs et sont utilisées pour adapter dynamiquement α et β. Nous proposons deux variantes qui diffèrent par la granularité de l'apprentissage : dans un cas l'apprentissage est global à l'ensemble des variables du problème, dans l'autre, il est spécifique à chaque variable. Nous évaluons et comparons expérimentalement ces deux méthodes dans le cadre d'une mise en œuvre de la méthode ACO pour résoudre des problèmes de satisfaction de contraintes.

Abstract

We introduce two reactive frameworks for dynamically adapting some parameters of an Ant Colony Optimization (ACO) algorithm. Both reactive frameworks use ACO to adapt parameters : pheromone trails are associated with parameter values ; these pheromone trails represent the learnt desirability of using parameter values and are used to dynamically set parameters in a probabilistic way. The two frameworks differ in the granularity of parameter learning. We experimentally evaluate these two frameworks on an ACO algorithm for solving constraint satisfaction problems.

1 Introduction

L'Optimisation par Colonies de Fourmis —Ant Colony Optimization / ACO— a démontré sont intérêt pour résoudre de nombreux problèmes combinatoires [4]. Par contre, la résolution d'un problème particulier avec ACO (de même que dans le cas d'autres méta-heuristiques) nécessite la résolution d'un compromis entre deux objectifs contradictoires : d'un coté, il est nécessaire d'intensifier la recherche de solution autour des zones les plus prometteuses alors que de l'autre, il est nécessaire de diversifier la recherche pour découvrir de nouvelles zones de l'espace de recherche qui pourraient s'avérer porteuses de meilleures solutions. C'est en réglant des paramètres de l'algorithme que l'on contrôle le comportement de la recherche vis-à-vis de ce double objectif d'intensification et de diversification (également nommés exploitation et exploration).

Le réglage de ces paramètres est bien entendu un problème délicat car il faut choisir entre deux tendances contradictoires : si l'on choisit des valeurs de paramètres qui privilégient la diversification, la qualité de la solution obtenue est souvent très bonne, mais au prix d'une convergence plus lente, et donc d'un temps de calcul plus long. Si par contre on choisit des valeurs qui favorisent l'intensification, l'algorithme sera amené à trouver de bonnes solutions plus rapidement, mais souvent sans converger vers la ou les meilleures solutions ; il convergera vers des valeurs sous-optimales. De ce fait, les valeurs optimales des paramètres dépendent à la fois de l'instance du problème à traiter et du temps alloué à sa résolution. De surcroit, la réso-

lution n'étant pas nécessairement homogène, il peut s'avérer utile de changer les valeurs des paramétres en cours de résolution afin de s'adapter au mieux aux caractéristiques locales de l'espace de recherche en cours d'exploration.

Pour améliorer le processus de recherche vis-à-vis de la dualité intensification/diversification, Battiti et al [2] ont proposé d'exploiter l'historique de la recherche pour adapter automatiquement et dynamiquement les valeurs de paramètres, donnant ainsi naissance aux approches dites "réactives".

Dans ce papier, nous introduisons une méthode réactive permettant à ACO d'adapter dynamiquement certains de ses paramètres pendant la recherche de solution. Cette adaptation dynamique est réalisée avec ACO lui-même : des traces de phéromone sont associées aux valeurs des paramètres ; elles représentent l'intérêt appris d'utiliser les valeurs des paramètres concernés et sont utilisées dynamiquement pour les affecter de façon probabiliste. Notre approche est évaluée expérimentalement dans le cadre de l'application de ACO pour résoudre des problèmes de satisfaction de contraintes (CSP) dont l'essence est de trouver une affectation de valeurs à des variables tout en respectant les contraintes auxquelles elles sont soumises.

L'article est organisé comme suit. Nous rappelons dans le chapitre 2 les principes de base des deux modèles que nous mettons en œuvre : les CSP et ACO. Le troisième chapitre est consacré à la description de la façon dont nous utilisons ACO pour adapter certains des paramètres pendant la recherche de la solution. Nous introduisons deux cadres réactifs pour ACO : dans le premier, les valeurs de paramètres sont fixées pour la durée de la construction de chaque solution, alors que dans le second, les paramètres sont associés à chaque variable —ils sont alors modifiés dynamiquement pendant la construction de chacune des solutions. Nous exposons au chapitre quatre l'évaluation expérimentale et la comparaison de ces deux variantes. Le dernier chapitre nous donne l'occasion de conclure en présentant des travaux proches ainsi que les évolutions que nous prévoyons d'explorer.

2 Contexte

2.1 Problèmes de satisfaction de contraintes (CSP)

Un *CSP* [12] est défini par un triplet (X, D, C) tel que X est un ensemble fini de variables, D est une fonction qui à chaque variable $X_i \in X$ associe un domaine $D(X_i)$, et C est un ensemble de contraintes, i.e., des relations entre variables qui restreignent l'ensemble des valeurs pouvant être affectées simultanément à ces variables.

Une *affectation*, notée $\mathcal{A} = \{<X_1, v_1>, .., <X_k, v_k>\}$, est un ensemble de couples variable/valeur tels que toutes les variables dans $\mathcal{A}$ sont différentes et chaque valeur appartient au domaine de la variable associée. Cette affectation correspond à l'affectation simultanée des valeurs $v_1, \ldots, v_k$ aux variables $X_1, \ldots, X_k$. Une affectation $\mathcal{A}$ est *partielle* s'il existe des variables de X qui ne sont pas affectées dans $\mathcal{A}$; elle est *complète* si toutes les variables sont affectées.

Le coût d'une affectation $\mathcal{A}$, dénoté par $cost(\mathcal{A})$, est défini par le nombre de contraintes qui sont violées par $\mathcal{A}$. Une *solution d'un CSP* (X, D, C) est une affectation complète de toutes les variables de X, qui satisfait toutes les contraintes dans C, c'est-à-dire, une affectation complète de coût nul.

De nombreux CSP réels, représentant des problèmes concrets, sont sur-contraints de sorte qu'il est impossible de trouver une affectation satisfaisant toutes les contraintes. Pour prendre en compte cette particularité, le cadre des CSP a été généralisé en introduisant la notion de *maxCSP* [5]. Dans ce cas, l'objectif n'est plus de trouver une solution consistante, mais de trouver une affectation qui maximise le nombre de contraintes respectées. La *solution optimale d'un maxCSP* est une affectation complète dont le coût est minimal.

2.2 Optimisation par Colonies de Fourmis (ACO)

La métaheuristique ACO [4] a été appliquée avec succès à un grand nombre de problèmes d'optimisation combinatoire tels que les problèmes de voyageurs de commerce, [3], les problèmes d'affectation quadratique ou bien les problèmes d'ordonnancement de voitures. Le principe de base est de construire des solutions d'une façon gloutonne aléatoire et de tirer parti de la connaissance acquise pour améliorer progressivement les solutions construites. Plus précisément, à chaque cycle, chaque fourmi construit une solution à partir d'une solution vide en ajoutant progressivement des *composants de solution* jusqu'à ce que la solution soit complète. A chaque itération de cette construction, le prochain composant de solution à ajouter est sélectionné relativement à une probabilité qui dépend d'un facteur phéromonal et d'un facteur heuristique :

- le facteur phéromonal reflète l'expérience de la colonie vis-à-vis de la sélection des composants. Il est défini relativement aux traces de phéromone associées aux composants de solutions. Ces traces de phéromone sont *renforcées* lorsque les composants de solutions correspondants on été sélectionnés dans de bonnes solutions. Toutefois, afin de permettre aux fourmis d'oublier progressivement les expériences passées, un mécanisme d'oubli —dual de la mémorisation— atténue les traces anciennes par *évaporation* à la fin de chaque cycle ;
- le facteur heuristique évalue l'intérêt de sélectionner les composants par rapport à la fonction d'objectif.

Ces deux facteurs sont respectivement pondérés par deux paramètres clefs de ACO : α et β. ACO intègre ainsi les deux connaissances complémentaires essentielles à la résolution efficace d'un problème : le facteur phéromonal représente la connaissance progressivement accumulée sur l'instance de problème en cours de résolution tandis que le facteur heuristique représente la connaissance sur la classe de problème dont l'instance est un cas particulier.

En plus de α et β, un algorithme ACO est également paramétré par

- le nombre de fourmis, *nbAnts*, qui détermine le nombre de solutions construites à chaque cycle ;
- le taux d'évaporation, $\rho \in]0;1[$, qui est utilisé à chaque cycle pour multiplier les traces de phéromone par $(1-\rho)$ afin de les atténuer à la fin de chaque cycle ;
- les bornes supérieures et inférieures, τ_{min} et τ_{max}, qui sont utilisées pour borner les traces de phéromone (si l'on considère le système MAX-MIN Ant System [11]).

Le cadre réactif proposé dans cet article vise à adapter α et β qui ont une très forte influence sur le processus de construction de solution.

La valeur du facteur phéromonal α est fondamentale pour articuler l'intensification et la diversification. En effet, plus α est grand, plus la recherche est intensifiée autour des composants de solution portant d'importantes traces de phéromone, i.e., autour des composants qui sont intervenus dans la construction des meilleures solutions passées. En particulier, nous avons montré dans [10] que le réglage du paramètre α doit trouver un compromis entre deux tendances. D'un coté, si l'on limite l'influence de la phéromone par une pondération faible, la qualité de la solution obtenue sera meilleure, mais il faudra plus de temps pour converger vers cette valeur. D'un autre coté, si l'on augmente l'influence du facteur phéromonal en augmentant la valeur du paramètre α, les fourmis trouvent de meilleures solutions pendant les premiers cycles, mais après quelques centaines de cycles, elles stagneront autour d'optima locaux et ne seront pas capables de trouver de meilleures solutions.

Le poids du facteur heuristique β détermine la *gloutonnerie* du processus de recherche ; ses valeurs les plus pertinentes dépendent également de l'instance à résoudre. En effet on constate que l'importance du facteur heuristique varie souvent d'une instance à l'autre. De surcroit, pour une instance donnée son importance peut évoluer pendant la recherche de la solution.

On remarque enfin que les valeur relatives de α et β sont bien sûr importantes, mais que leurs valeurs absolues le sont également —dans le cas contraire un seul paramètre aurait suffit.

Algorithm 1: Ant Solver

Input: A CSP (X, D, C) and a set of parameters $\{\alpha, \beta, \rho, \tau_{min}, \tau_{max}, nbAnts, maxCycles\}$
Output: A complete assignment for (X, D, C)

1 Initialize pheromone trails associated with (X, D, C) to τ_{max}
2 **repeat**
3 **foreach** k **in** $1..nbAnts$ **do**
4 Construct an assignment $\mathcal{A}_k$
5 Improve $\mathcal{A}_k$ by local search
6 Evaporate each pheromone trail by multiplying it by $(1-\rho)$
7 Reinforce pheromone trails of $\mathcal{A}_{best} = arg\min_{\mathcal{A}_k \in \{\mathcal{A}_1,\ldots,\mathcal{A}_{nbAnts}\}} cost(\mathcal{A}_k)$
8 **until** $cost(\mathcal{A}_i) = 0$ *for some* $i \in \{1..nbAnts\}$ **or** *maxCycles reached* ;
9 **return** the constructed assignment with the minimum cost

2.3 Résolution de maxCSP avec ACO

L'algorithme ACO considéré dans notre étude comparative s'appelle Ant Solver (AS) et est décrit dans l'algorithme 1. Nous rappelons rapidement ci-dessous son principe ; une description plus détaillée peut être trouvée dans [8, 9].

Traces de phéromone associées à un CSP (X, D, C) (ligne 1). Nous associons une trace de phéromone à chaque couple variable/valeur $\langle X_i, v\rangle$ tel que $X_i \in X$ et $v \in D(X_i)$. Intuitivement, cette trace de phéromone représente l'intérêt appris d'affecter la valeur v à la variable X_i. Ainsi qu'il est proposé dans [11], les traces de phéromone sont bornées entre τ_{min} et τ_{max}, et sont initialisées à τ_{max}.

Construction d'une affectation par une fourmi (ligne 4) : à chaque cycle (lignes 2-8), chaque fourmi construit une affectation : à partir d'une affectation vide $\mathcal{A} = \emptyset$, elle ajoute itérativement des couples variable/valeur à $\mathcal{A}$ jusqu'à ce que $\mathcal{A}$ soit complète. A chaque étape, pour choisir un couple variable/valeur, la fourmi choisit d'abord une variable X_j qui n'est pas encore affectée dans $\mathcal{A}$. Ce choix est réalisé avec l'heuristique `min-domain` : la fourmi choisit une variable qui a le plus petit nombre de valeurs consistantes vis-à-vis de l'affectation partielle $\mathcal{A}$ en cours de construction. Puis, la fourmi choisit une valeur $v \in D(X_j)$ à affecter à X_j selon la probabilité suivante :

$$p_{\mathcal{A}}(<X_j, v>) = \frac{[\tau(<X_j,v>)]^{\alpha} \cdot [\eta_{\mathcal{A}}(<X_j,v>)]^{\beta}}{\sum_{w \in D(X_j)} [\tau(<X_j,w>)]^{\alpha} \cdot [\eta_{\mathcal{A}}(<X_j,w>)]^{\beta}}$$

où

– $\tau(< X_j, v >)$ est la trace de phéromone liée à $(< X_j, v>)$,
– $\eta_{\mathcal{A}}(< X_j, v >)$ est le facteur heuristique et est inversement proportionnel au nombre de nouvelles contraintes violées par l'affectation de la valeur v à la variable X_j, c.-à-d., $\eta_{\mathcal{A}}(< X_j, v >)) = 1/(1 + cost(\mathcal{A} \cup \{<X_j, v>\}) - cost(\mathcal{A}))$,
– α et β sont les paramètres qui déterminent les poids relatifs des facteurs.

Amélioration locale des affectations (ligne 5) : lorsqu'une affectation complète a été construite par une fourmi, elle est améliorée en exécutant une recherche locale, c.-à-d., en changeant itérativement quelques affectations variable/valeur. Différentes heuristiques peuvent être employées pour choisir la variable à réparer et la nouvelle valeur à lui affecter. Pour toutes les expériences rapportées ci-après, nous avons employé l'heuristique `min-conflicts` [6] : nous choisissons aléatoirement une variable impliquée dans une contrainte violée, et nous affectons cette variable avec la valeur qui minimise le nombre de contraintes violées. De telles améliorations locales sont réitérées jusqu'à atteindre une solution localement optimale qui ne peut pas être améliorée en modifiant une affectation de variable.

Mise à jour des traces de phéromone (lignes 6-7) : lorsque chacune des fourmis de la colonie a construit une affectation, et l'a améliorée par recherche locale, la quantité de phéromone associée à chaque couple variable/valeur est mise à jour selon la méta-heuristique ACO. En premier lieu, toutes les traces de phéromone sont uniformément diminuées (ligne 6) afin de simuler la forme d'évaporation qui permet aux fourmis d'oublier progressivement les constructions passées. Puis, la phéromone est ajoutée sur chaque couple variable/valeur appartenant à la meilleure affectation du cycle, $\mathcal{A}_{best}$ (ligne 7) afin d'attirer les fourmis avec plus de force vers la zone correspondante de l'espace de recherche. La quantité de phéromone déposée est inversement proportionnelle au nombre de contraintes violées dans $\mathcal{A}_{best}$, i.e., $1/cost(\mathcal{A}_{best})$.

3 Utilisation d'ACO pour adapter dynamiquement α et β

Nous proposons d'*employer ACO pour adapter dynamiquement les valeurs* de α et de β. En particulier, nous proposons et comparons dans ce cadre deux méthodes différents. Dans le premier cas, appelé AS($\mathcal{GPL}$) et décrit dans 3.1, les valeurs de α et de β sont fixées au début de la construction d'une solution et sont *adaptées après chaque cycle*, lorsque toutes les fourmis ont construit une solution. Dans la deuxième variante, appelé AS($\mathcal{DPL}$) et décrite dans 3.2, les valeurs de α et de β sont définies pour chaque variable de sorte qu'elles *changent pendant la construction d'une solution*. Ces deux variantes sont expérimentalement évaluées dans 4.

3.1 Description de AS($\mathcal{GPL}$)

AS($\mathcal{GPL}$) (Ant Solver with Global Parameter Learning) suit essentiellement l'algorithme 1 mais intègre de nouveaux dispositifs pour adapter dynamiquement α et β. Par conséquent, α et β ne sont plus à fournir comme paramètres en entrée de l'algorithme, mais leurs valeurs sont choisies avec la méta-heuristique ACO à chaque cycle [1].

Paramétres de AS($\mathcal{GPL}$). En plus des paramètres de Ant Solver, c.-à-d., le nombre de cycles *nbCycles*, le nombre de fourmis *nbAnts*, le taux d'évaporation ρ, et les bornes minimales et maximales des traces de phéromone τ_{min} et τ_{max}, AS($\mathcal{GPL}$) est paramétré par un ensemble de nouveaux paramètres permettant d'adapter dynamiquement α et β, i.e.,
– deux ensembles de valeurs $\mathcal{I}\alpha$ et $\mathcal{I}_\beta$ qui contiennent les ensembles de valeurs qui peuvent être respectivement considérées pour l'affectation de α et β ;
– les bornes minimale et maximale de la quantité de phéromone, $\tau_{min_{\alpha\beta}}$ et $\tau_{max_{\alpha\beta}}$;
– le taux d'évaporation $\rho_{\alpha\beta}$.

Il est à noter que notre cadre réactif suppose que α et β prennent leurs valeurs dans deux ensembles discrets $\mathcal{I}_\alpha$ et $\mathcal{I}_\beta$ qui doivent être connus *a priori*. Ces deux ensembles doivent contenir de bonnes valeurs, *c.-à-d.*, celles qui permettent à Ant Solver de trouver les meilleurs résultats pour chaque exemple. Comme discuté dans la section 4, nous proposons de choisir les valeurs de $\mathcal{I}_\alpha$ et $\mathcal{I}_\beta$ en lançant Ant Solver avec différentes affectations pour α et β sur un ensemble représentatif d'instances, puis d'initialiser $\mathcal{I}_\alpha$ et $\mathcal{I}_\beta$ avec les ensembles des valeurs qui ont permis à Ant Solver de trouver les meilleurs résultats sur ces instances.

Structure de phéromone. Nous associons une trace de phéromone $\tau_\alpha(i)$ à chaque valeur $i \in \mathcal{I}_\alpha$ et une trace de phéromone $\tau_\beta(j)$ à chaque valeur $j \in \mathcal{I}_\beta$. Intuitivement, ces traces de phéromone représentent l'intérêt appris d'affecter respectivement α et β à i et à j. Pendant le processus de recherche, ces traces de phéromone sont maintenues entre les deux limites $\tau_{min_{\alpha\beta}}$ et $\tau_{max_{\alpha\beta}}$. Au début du processus de recherche, elles sont initialisées à $\tau_{max_{\alpha\beta}}$.

1. Nous avons comparé expérimentalement deux variantes de ce cadre réactif : une première variante où les valeurs sont choisies au début de chaque cycle (entre les lignes 2 et 3) de sorte que chaque fourmi considère les mêmes valeurs pendant le cycle, et une deuxième variante où les valeurs sont choisies par des fourmis avant de construire une affectation (entre les lignes 3 et 4). Les deux variantes obtiennent des résultats qui ne sont pas sensiblement différents. Par conséquent, nous considérons seulement la première variante qui est décrite dans cette section.

Choix des valeurs pour α et β. A chaque cycle, (i.e., entre les lignes 2 et 3 de l'algorithme 1), α (resp. β) est affecté en choisissant $i \in \mathcal{I}_\alpha$ (resp. $i \in \mathcal{I}_\beta$) relativement à une probabilité $p_\alpha(i)$ (resp. $p_\beta(i)$) qui est proportionelle à la quantité de phéromone déposée sur i, i.e.,

$$p_\alpha(i) = \frac{\tau_\alpha(i)}{\sum_{j \in \mathcal{I}_\alpha} \tau_\alpha(j)} \quad (\text{resp. } p_\beta(i) = \frac{\tau_\beta(i)}{\sum_{j \in \mathcal{I}_\beta} \tau_\beta(j)})$$

Mise à jour des traces de phéromone. Les traces de phéromone associées à α et β sont mises à jour à chaque cycle, entre les lignes 7 et 8 de l'algorithme 1. D'abord, chaque trace de pheromone $\tau_\alpha(i)$ (resp. $\tau_\beta(i)$) est évaporée en la multipliant par $(1 - \rho_{\alpha\beta})$. Ensuite la trace de pheromone associée à α (resp. β) est renforcée. La quantité de pheromone déposée sur $\tau_\alpha(\alpha)$ (resp. $\tau_\beta(\beta)$) est inversement proportionnelle au nombre de contraintes violées $\mathcal{A}_{best}$ par la meilleure affectation construite pendant le cycle. Ainsi, les valeurs de α et β qui ont permis aux fourmis de construire les meilleures affectations recevront les plus grandes quantité de phéromone.

3.2 Description de AS($\mathcal{DPL}$)

Le cadre réactif décrit à la section précédente adapte dynamiquement α et β à chaque cycle, mais il considère la même affectation pour toutes les décisions élémentaires d'un même cycle. Nous décrivons maintenant une autre variante réactive nommée AS($\mathcal{DPL}$) (Ant Solver with Distributed Parameter Learning). Le principe est de choisir de nouvelles valeurs pour α et β à chacune des étapes de la construction d'une solution, c.-à-d., chaque fois qu'une fourmi doit choisir une valeur pour une variable. Le but est d'ajuster l'affectation de α et de β pour chaque variable du CSP.

Paramètres de AS($\mathcal{DPL}$). Les paramètres de AS($\mathcal{DPL}$) sont les mêmes que ceux de AS($\mathcal{GPL}$).

Structure phéromonale Nous associons une trace de phéromone $\tau_\alpha(X_k, i)$ (resp. $\tau_\beta(X_k, i)$) à chaque variable $X_k \in X$ et chaque valeur $i \in \mathcal{I}_\alpha$ (resp. $i \in \mathcal{I}_\beta$). Intuitivement ces traces de pheromone représentent l'intérêt appris d'affecter la valeur i à α (resp. β) lorsqu'on choisit une valeur pour la variable X_k. Pendant le processus de recherche de solution ces traces de pheromone sont maintenues entre les deux bornes $\tau_{min_{\alpha\beta}}$ et $\tau_{max_{\alpha\beta}}$; elles sont initialisées à $\tau_{max_{\alpha\beta}}$.

Choix des valeurs pour α et β. A chaque étape de la construction d'une affectation, avant de choisir une valeur v pour une variable X_k, α (resp. β) est positionné en choisissant une valeur $i \in \mathcal{I}_\alpha$ (resp. $i \in \mathcal{I}_\beta$) en fonction de la probabilité $p_\alpha(X_k, i)$ (resp. $p_\beta(X_k, i)$) qui est proportionnelle à la quantité de phéromone associée à i pour X_k, i.e.,

$$\begin{cases} p_\alpha(X_k, i) = \dfrac{\tau_\alpha(X_k, i)}{\sum_{j \in \mathcal{I}_\alpha} \tau_\alpha(X_k, j)} \\ \\ p_\beta(X_k, i) = \dfrac{\tau_\beta(X_k, i)}{\sum_{j \in \mathcal{I}_\beta} \tau_\beta(X_k, j)}) \end{cases} \quad (1)$$

Mise à jour des traces de phéromone. Les traces de phéromone associées à α et β sont mises à jour à chaque cycle : entre les lignes 7 et 8 de l'algorithme 1. D'abord chaque trace de phéromone $\tau_\alpha(X_k, i)$ (resp. $\tau_\beta(X_k, i)$) est évaporée en la multipliant par $(1 - \rho_{\alpha\beta})$. Ensuite, une quantité de pheromone est déposée sur les traces associées aux valeurs α et β qui ont été utilisées pour construire la meilleure affectation du cycle ($\mathcal{A}_{best}$) : pour chaque variable $X_k \in X$, si α (resp. β) a été affecté à i pour choisir la valeur affectée à X_k lors de la construction de $\mathcal{A}_{best}$, alors, $\tau_\alpha(X_k, i)$ (resp. $\tau_\beta(X_k, i)$) est incrémenté de $1/cost(\mathcal{A}_{best})$.

4 Résultats expérimentaux

4.1 Instances considérées

Nous illustrons notre ACO réactif sur le benchmark de maxCSP qui a été utilisé pour la compétition [13] CSP 2006, en considérant les 686 instances binaires de maxCSP définies en extension. Parmi ces 686 instances, 641 sont résolues à l'optimal [2], à la fois par la version statique de Ant Solver et par les deux versions réactives, les temps de réponses ne diffèrant pas significativement. Nous avons donc concentré notre expérimentation de la section 4.3 sur les 25 instances les plus difficiles. Parmi ces 25 instances, nous avons sélectionné 10 instances représentatives dont les caractéristiques sont décrites dans la table 1. Nous développons les résultats expérimentaux pour toutes les instances du benchmark de la compétition lorsque nous comparons notre modèle d'ACO réactif avec les meilleurs solveurs de la compétition.

4.2 Contexte d'expérimentation

Pour régler les paramètres de Ant Solver nous l'avons fait tourner en faisant varier les paramètres sur un sous-ensemble représentatif de 100 instances choisies parmi les 686 instances de la compétition. Ces 100 instances contiennent les 25 plus difficiles. Puis nous avons sélectionné le paramétrage avec lequel Ant Solver trouve en

2. Pour la plupart de ces instances, la meilleure solution est connue, cependant pour quelques instances, la solution optimale ne l'est pas. Pour ces instances, nous avons considéré la meilleure solution connue.

Nb	Nom	X	D	C	B
1	brock-400-1	401	2	20477	378
2	brock-400-2	401	2	20414	378
3	mann-a27	379	2	1080	252
4	san-400-0.5-1	401	2	40300	392
5	rand-2-40-16-250-350-30	40	16	250	1
6	rand-2-40-25-180-500-0	40	25	180	1
7	rand-2-40-40-135-650-10	40	40	135	1
8	rand-2-40-40-135-650-22	40	40	135	1

TABLE 1 – Pour chaque instance, Nom, X, D, C, et B indiquent respectivement le nom, le nombre de variables, la taille des domaines des variables, le nombre de contraintes, et le nombre de contraintes violées par la meilleure solution trouvée lors de la compétition de 2006.

moyenne les meilleurs résultats, i.e., $\alpha = 2$, $\beta = 8$, $\rho = 0.01$, $\tau_{min} = 0.1$, $\tau_{max} = 10$, and $nbAnts = 15$. Nous avons limité le nombre de cycles à 10000, mais on constate que le nombre de cycles réellement nécessaire à l'obtention de la meilleure solution est souvent très inférieur. Dans cette section, AS(Static) se réfère à Ant Solver utilisé avec ce paramétrage.

Nous avons également réglé α et β séparément pour chaque instance (tout en conservant inchangées les valeurs des autres paramètres). Dans cette section, AS(Tuned) se réfère à Ant Solver dont les paramètres statiques sont affectés avec la meilleure valeur pour l'instance considérée.

Pour les deux variantes réactives de Ant Solver (AS($\mathcal{GPL}$) et AS($\mathcal{DPL}$)), les paramètres non appris gardent les mêmes valeurs que pour $AS(Tuned)$ et AS $(Static)$, i.e., $\rho = 0.01$, $\tau_{min} = 0.1$, $\tau_{max} = 10$, et $nbAnts = 15$.

Pour les nouveaux paramètres, qui ont été introduits pour adapter dynamiquement α et β, nous avons positionné $\mathcal{I}_\alpha$ et $\mathcal{I}_\beta$ aux valeurs qui donnent de bons résultats pour Static Ant Solver, i.e., $\mathcal{I}_\alpha = \{0, 1, 2\}$ et $\mathcal{I}_\beta = \{0, 1, 2, 3, 4, 5, 6, 7, 8\}$. Quant au taux d'évaporation, et aux bornes supérieures et inférieures des traces de phéromone, nous avons utilisé les même valeurs que pour AS(Static), i.e., $\rho_{\alpha\beta} = 0.01$, $\tau_{min_{\alpha\beta}} = 0.1$, $\tau_{max_{\alpha\beta}} = 10$.

4.3 Comparaison expérimentale de AS(Tuned), AS(Static), AS($\mathcal{GPL}$) et AS($\mathcal{DPL}$)

La table 3 indique la meilleure affectation pour α et β qui a été utilisée pour éxecuter AS(Tuned). Elle montre que cette meilleure affectation est clairement différente d'une instance à l'autre. Nous avons également observé qu'à la fin du processus de recherche de AS($\mathcal{GPL}$), les traces de phéromone utilisée pour les affecter portent des valeurs différentes d'une instance à l'autre. Ceci est particulièrement marqué pour le paramètre β, ce qui indique que la pertinence du facteur heuristique dépend de l'instance considérée.

	1	2	3	4	5	6	7	8
AS($\mathcal{DPL}$)/AS($\mathcal{GPL}$)	=	=	=	=	$>$	$>$	=	$>$
AS($\mathcal{DPL}$)/AS(Static)	=	$>$	$>$	$>$	=	$>$	=	=
AS($\mathcal{DPL}$)/AS(Tuned)	=	$>$	=	=	=	$>$	$<$	$<$
AS($\mathcal{GPL}$)/AS(Static)	=	$>$	$>$	$>$	$<$	=	$<$	=
AS($\mathcal{GPL}$)/AS(Tuned)	=	$>$	=	=	$<$	=	$<$	$<$
AS(Static)/AS(Tuned)	=	=	$<$	$<$	=	=	$<$	$<$

TABLE 2 – Résultats des tests de pertinence statistique : chaque ligne compare deux variantes X/Y et donne pour chaque instance les résultats des tests sur 50 exécutions, *i.e.*, = (resp. $<$ et $>$) si X n'est pas significativement différent de Y (resp. est significativement moins bon ou meilleur que Y).

rée.

La table 3 compare également le nombre de contraintes violées dans la meilleure solution trouvée après 10000 cycles pour les quatre variantes de Ant Solver. Comme les différences entre ces variantes sont plutôt faibles pour certaines instances, nous avons réalisé des tests de pertinence statistique. La table 2 fournit les résultats de ces tests. Elle met en évidence que les variantes réactives sont toujours au moins aussi performantes que AS(Static), excepté pour les instances 5 et 7 qui sont mieux résolues par AS(Static) que par AS($\mathcal{GPL}$). Elle nous indique également que les deux variantes réactives sont au moins au même niveau de performance que AS(Tuned), et qu'elles le surpassent même pour de nombreuses instances – en fait, toutes sauf 2 pour AS($\mathcal{DPL}$) et toutes sauf 3 pour AS($\mathcal{GPL}$)). Enfin la table indique également que AS($\mathcal{DPL}$) n'est pas significativement différent de AS($\mathcal{GPL}$) pour 5 instances, et le dépasse pour 3 instances.

La table 4 compare le nombre de cycles et le temps CPU nécessaire à l'obtention de la meilleure solution. Nous notons d'abord que la surcharge en temps de calcul induite par la mise en œuvre de notre méthode réactive n'est pas significative ; les variantes de Ant Solver mettent un temps comparable pour réaliser un cycle de calcul d'une instance à l'autre, mais également d'une variante de Ant Solver à l'autre. En particulier, AS(Static) converge souvent plus rapidement que AS(Tuned).

Pour comparer les quatre variantes de Ant Solver pendant l'ensemble de la recherche de solution, et non seulement à la fin des 10000 cycles, la figure 1 représente l'évolution du pourcentage d'exécutions qui ont trouvées la solution optimale, en fonction du nombre de cycles [3]. La figure indique que AS(Static) peut trouver la solution optimale pour plus de la moitié des exécutions avant le 2000^{ieme} cycle. Cependant, après 2000 cycles, le pourcentage des exécutions résolues à l'optimalité ne croit que

3. Comme la preuve d'optimalité n'a pas été faite pour toutes les instances, nous considérons la meilleure solution trouvée pour les solutions des instances dont l'optimalité n'a pas été prouvée.

	AS(Tuned)				AS(Static)		AS($\mathcal{GPL}$)		AS($\mathcal{DPL}$)	
Nb	#const	(sdv)	α	β	#const	(sdv)	#const	(sdv)	#const	(sdv)
1	374.84	(0.7)	1	6	374.92	(0.39)	**374.**	(1.01)	**374.**	(1.01)
2	373.12	(0.26)	1	5	374.68	(1.09)	**371.32**	(1.09)	371.48	(1.31)
3	253.88	(0.26)	1	6	254.62	(0.49)	**253.74**	(0.44)	253.96	(0.28)
4	387.2	(0.11)	1	8	388.04	(1.77)	**387.**	(0.)	**387.**	(0.)
5	**1.**	(0.)	2	8	**1.**	(0.)	1.02	(0.14)	**1.**	(0.)
6	1.02	(0.02)	2	6	1.02	(0.14)	1.04	(0.19)	**1.**	(0.)
7	**1.**	(0.)	1	6	1.12	(0.32)	1.66	(0.47)	1.48	(0.5)
8	**1.**	(0.)	1	5	1.08	(0.27)	1.12	(0.32)	1.08	(0.27)

TABLE 3 – Comparaison expérimentale des meilleures solutions trouvées. Pour chaque variante de Ant Solver considérée, chaque ligne indique le nombre de contraintes violées dans la meilleure solution (moyenne et écart-type sur 50 exécutions). Pour AS(Tuned), nous donnons également les valeurs de α et β qui ont été utilisées.

	AS(Tuned)			AS(Static)			AS($\mathcal{GPL}$)			AS($\mathcal{DPL}$)		
	cycles		time	cycles		time	cycles		time	cycles		time
Nb	avg	(sdv)	avg	avg	(sdv)	avg	avg	(sdv)	avg	avg	(sdv)	avg
1	44	(7)	4	30	(4)	2	2717	(516)	98	2501	(491)	93
2	2247	(477)	140	323	(194)	12	2668	(463)	96	3322	(415)	125
3	2193	(309)	542	1146	(335)	160	2714	(432)	399	2204	(295)	328
4	710	(213)	123	347	(174)	39	316	(38)	34	112	(14)	12
5	394	(32)	5	394	(32)	4	379	(44)	5	412	(37)	5
6	476	(19)	26	606	(23)	13	507	(49)	18	579	(23)	15
7	2436	(166)	160	1092	(152)	31	736	(126)	39	1557	(266)	52
8	1944	(120)	140	884	(65)	29	1302	(286)	66	1977	(252)	87

TABLE 4 – comparaison expérimentale du nombre de cycles (moyenne et écart-type sur 50 exécutions) et du temps CPU exprimé en secondes (moyenne sur 50 exécutions) nécessaires pour atteindre la meilleure solution.

faiblement. AS(Tuned) affiche un comportement assez différent : il nécessite plus de cycles pour converger de sorte qu'il trouve moins souvent la solution optimale au début du processus de recherche ; cependant, il présente des performances nettement supérieures à AS(Static) à la fin des 10000 cycles. Considèrons par exemple les instances 2, 3 et 8 : la table 3 nous montre que AS(Tuned) à besoin de plus de cycles que AS(Static) pour les résoudre.

La figure 1 nous indique également que AS($\mathcal{GPL}$) présente de meilleures performances que les trois autres variantes pendant les 2000 premier cycles, alors qu'après 2000 cycles AS($\mathcal{GPL}$), AS($\mathcal{DPL}$) et AS(Tuned) sont assez proches et ont toutes de meilleures performances que AS(Static). Enfin, à la fin du processus de recherche, AS($\mathcal{DPL}$) est légèrement meilleur que AS($\mathcal{GPL}$) qui lui-même est légèrement meilleur que AS(Tuned).

La figure 2 montre l'évolution du pourcentage d'exécutions qui sont proches de l'optimalité, *i.e.*, les solutions optimales ou les solutions qui violent une contrainte de plus que la solution optimale. Elle montre que AS($\mathcal{GPL}$) trouve les solutions quasi-optimales plus rapidement que AS($\mathcal{DPL}$), qui lui-même affiche une meilleure performance que les variantes statiques de Ant Solver. AS(Static) est meilleur que AS(Tuned) au début de la recherche de solution, mais il est généralement dépassé par AS(Tuned) au-delà des 1000 cycles.

4.4 Comparaison expérimentale de AS($\mathcal{DPL}$) avec les solvers de la compétition

Nous comparons maintenant AS($\mathcal{DPL}$) avec les solveurs de maxCSP de la compétition 2006. Il y avait neuf solveurs, parmi lesquels huit sont basés sur des approches complètes, et un est basé sur une méthode de recherche locale incomplète. Pour la compétition, chaque solveur a bénéficié d'une durée allant jusqu'à 40 minutes sur un 3GHz Intel Xeon (voir [13] pour plus de détails). Pour chaque exemple, nous avons comparé AS($\mathcal{DPL}$) avec le meilleur résultat trouvé pendant la compétition (par l'un des 9 solvers). Nous avons également limité AS($\mathcal{DPL}$) à 40 minutes, mais il a été exécuté sur un ordinateur moins puissant (Intel P4 Dual Core à 1.7 GHz). Nous ne rapportons pas les temps CPU car ils ont été obtenus sur différents ordinateurs. Le but ici est d'évaluer la qualité des solutions

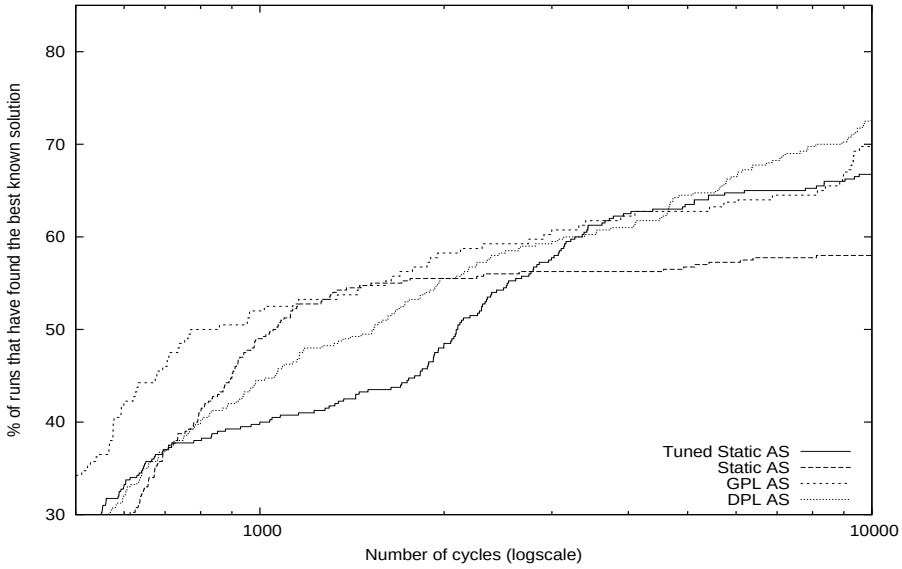


FIGURE 1 – Evolution du pourcentage d'exécutions qui ont trouvé la solution optimale par rapport au nombre de cycles.

				Competition		AS($\mathcal{DPL}$)	
Bench	#I	#C	$\sum$ best known	$\sum cost$	#I^{best}	$\sum cost$	#I^{best}
brock	4	56381	1111	1123	2	**1111**	**4**
hamming	4	14944	460	463	1	**460**	**4**
mann	2	1197	281	**281**	**2**	283	1
p-hat	3	312249	1472	1475	1	**1472**	**3**
san	3	48660	687	692	2	**687**	**3**
sanr	1	6232	182	183	0	**182**	**1**
dsjc	1	736	19	20	0	**19**	**1**
le	2	11428	2869	2925	1	**2869**	**2**
graphw	6	16993	416	420	4	**416**	**6**
scenw	27	29707	809	904	25	**809**	**27**
tightness0.5	15	2700	15	**15**	**15**	16	14
tightness0.65	15	2025	15	**15**	**15**	18	12
tightness0.8	15	1545	21	**22**	**13**	25	10
tightness0.9	15	1260	26	**30**	**11**	31	10

TABLE 5 – Comparaison expérimentale de AS($\mathcal{DPL}$) avec les solveurs de la compétition 2006. Chaque ligne indique le nom du benchmark, le nombre d'instances dans ce benchmark (#I), le nombre total de contraintes dans ces instances (#C), et le nombre de contraintes violées en considérant, pour chaque instance, le meilleur résultat trouvé ($\sum$ best known). Puis, nous donnons les meilleurs résultats obtenus pendant la compétition : pour chaque instance, nous avons considéré le meilleur résultat des 9 solveurs de la compétition et nous donnons le nombre de contraintes qui sont violées ($\sum cost$) suivi du nombre d'instances pour lesquelles la meilleure solution a été trouvée (#I ^{best}). Enfin, nous donnons les résultats obtenus avec AS($\mathcal{DPL}$) : le nombre de contraintes violées ($\sum cost$) suivi du nombre d'instances pour lesquelles la meilleure solution connue a été trouvée (#I^{bests}).

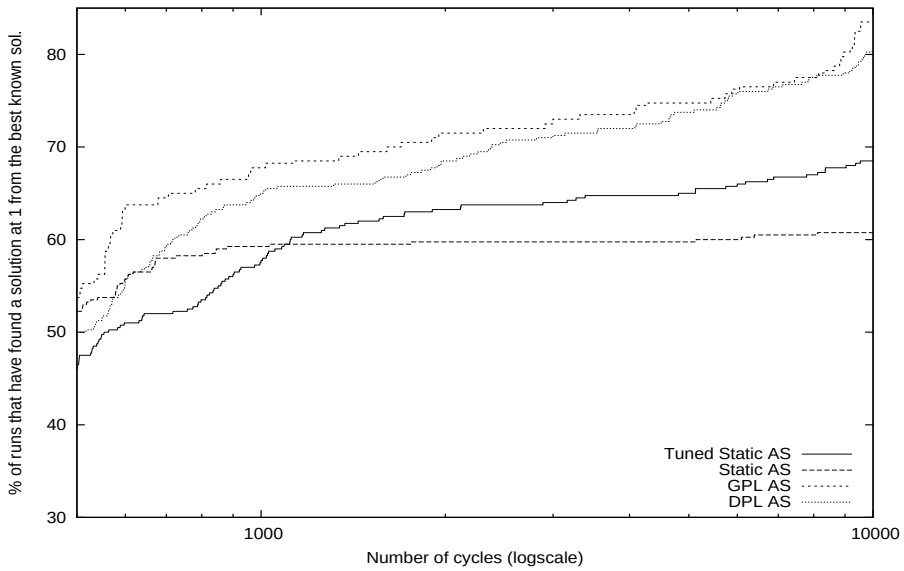


FIGURE 2 – Evolution du pourcentage d'exécutions qui ont trouvé une solution optimale, ou violant une contrainte de plus que la solution optimale, en fonction du nombre de cycles.

trouvées par AS($\mathcal{DPL}$).

Cette comparaison a été faite sur les 686 instances binaires définis en extension. Ces instances ont été regroupés dans 45 benchmarks. Parmi ces 45 benchmarks, il y avait 31 benchmarks pour lesquels AS($\mathcal{DPL}$) et les meilleurs solveurs de la compétition ont trouvé les mêmes valeurs pour chaque instance. Par conséquent, nous ne montrons les résultats que pour les 14 benchmarks pour lesquels AS($\mathcal{DPL}$) et le meilleur solveur de la compétition ont obtenu des résultats différents (pour au moins une instance du benchmarks).

La table 5 donne les résultats pour ces 14 benchmarks. Elle montre que AS($\mathcal{DPL}$) a de meilleures performances que les meilleurs solvers de la compétition pour 9 benchmarks. Plus précisément, AS($\mathcal{DPL}$) a amélioré les meilleures solutions trouvées par un solveur de la compétition pour 19 instances. Par contre, il n'a pas trouvé la meilleure solution pour 15 instances ; parmi ces 15 instances, 14 appartiennent à la série `tightness*` qui apparait clairement plus difficile pour AS($\mathcal{DPL}$) (notons cependant qu'aucun solveur de la compétition n'a été capable de trouver la solution optimale pour 6 de ces instances).

5 Conclusion

Nous avons présenté deux variantes d'un formalisme réactif pour adapter automatiquement et dynamiquement les valeurs du poids α du facteur phéromonal et du poids β du facteur heuristique qui ont une forte influence sur l'articulation intensification/diversification de la recherche dans ACO. Le but est double : nous visons en premier lieu à libérer l'utilisateur du problème délicat du réglage de ces paramètres ; nous visons également à améliorer les résultats sur des instances difficiles.

Les premiers résultats expérimentaux sont très encourageants. En effet, dans la plupart des cas notre ACO réactif atteint les performances d'une variante statique dont le paramétrage a été spécialement réglé pour chaque instance ; il les surpasse même sur quelques instances.

Travaux comparables. Il y a de nombreux travaux relatifs aux approches réactives, qui adaptent dynamiquement des paramètres pendant le processus de recherche [2]. Plusieurs de ces approches réactives ont été proposées pour des approches de recherche locale, et sont à l'origine de la notion de recherche réactive. Par exemple, Battiti et Protasi ont proposé dans [1] d'employer l'information de rééchantillonnage afin d'adapter dynamiquement la longueur de la liste taboue dans une recherche taboue.

Il existe également d'autres approches réactives pour les algorithmes ACO. En particulier, Randall a proposé dans [7] d'utiliser ACO pour adapter dynamiquement des paramètres d'un algorithme ACO. Notre approche emprunte quelques caractéristiques à ce modèle ACO réactif. Ce-

pendant nous apprenons les paramètres à un niveau différent. En effet, dans [7] les paramètres sont appris au niveau des fourmis de sorte que chaque fourmi fait évoluer ses propres paramètres et considère le même paramétrage pendant une construction de solution. Dans ce modèle, chaque fourmi est considérée comme un agent autonome qui améliore ses performances au fur et à mesure qu'il acquiert de la connaissance sur le problème. Notre approche est différente, les paramètres sont appris au niveau de la colonie, de sorte que chaque fourmi emploie les mêmes traces de phéromone pour fixer ses paramètres. Par ailleurs, nous avons étudié et comparé deux variantes : dans la première, les mêmes paramètres sont employés pendant l'ensemble de la construction d'une solution, alors que dans la seconde, les paramètres sont associés à chaque variable. Nous avons montré que cette variante améliore réellement le processus de recherche sur certains instances, mettant en évidence que, lors de la résolution de problèmes de satisfaction de contraintes, la pertinence des facteurs heuristiques et de phéromone dépendent de la variable à assigner.

Futurs développements. Nous prévoyons d'évaluer notre cadre réactif sur d'autres mises en œuvre de ACO afin de qualifier sa généricité. En particulier, il sera intéressant de comparer les variantes réactives sur d'autres problèmes : pour certains problèmes tels que le Voyageur de Commerce, il est plus probable que l'association de paramètres à chaque composant de solution ne soit pas intéressante, tandis que sur d'autres problèmes, tels que le sac à dos multidimensionnel ou les problèmes d'ordonnancement, nous conjecturons que ceci devrait améliorer le processus de recherche.

Une limite de notre cadre réactif se situe dans le fait que l'espace de recherche pour les valeurs de paramètre doit être connu à l'avance et discrétisé. Comme précisé par un relecteur, il serait préférable de résoudre ce méta-problème en tant que tel, *i.e.*, un problème d'optimisation continue. Par conséquent, un développement futur abordera cet aspect.

En conclusion, nous prévoyons d'intégrer un cadre réactif pour adapter dynamiquement les autres paramètres, ρ, τ_{min}, et τ_{max} qui ont des dépendances fortes avec α et β. Ceci pourrait être fait, par exemple, en employant des indicateurs d'intensification/diversification, tels que les taux de similarité ou de rééchantillonnage.

Références

[1] R. Battiti and M. Protasi. Reactive local search for the maximum clique problem. *Algorithmica*, 29(4) :610–637, 2001.

[2] Roberto Battiti, Mauro Brunato, and Franco Mascia. *Reactive Search and Intelligent Optimization*. Operations research/Computer Science Interfaces. Springer Verlag, 2008. in press.

[3] M. Dorigo and L.M. Gambardella. Ant colony system : A cooperative learning approach to the traveling salesman problem. *IEEE Transactions on Evolutionary Computation*, 1(1) :53–66, 1997.

[4] M. Dorigo and T. Stuetzle. *Ant Colony Optimization*. MIT Press, 2004.

[5] E. Freuder and R. Wallace. Partial constraint satisfaction. *Artificial Intelligence*, 58 :21–70, 1992.

[6] S. Minton, M.D. Johnston, A.B. Philips, and P. Laird. Minimizing conflicts : a heuristic repair method for constraint satistaction and scheduling problems. *Artificial Intelligence*, 58 :161–205, 1992.

[7] Marcus Randall. Near parameter free ant colony optimisation. In *ANTS Workshop*, volume 3172 of *Lecture Notes in Computer Science*, pages 374–381. Springer, 2004.

[8] C. Solnon. Ants can solve constraint satisfaction problems. *IEEE Transactions on Evolutionary Computation*, 6(4) :347–357, 2002.

[9] C. Solnon and D. Bridge. An ant colony optimization meta-heuristic for subset selection problems. In *System Engineering using Particle Swarm Optimization*, pages 7–29. Nova Science, 2006.

[10] C. Solnon and S. Fenet. A study of aco capabilities for solving the maximum clique problem. *Journal of Heuristics*, 12(3) :155–180, 2006.

[11] T. Stützle and H.H. Hoos. $\mathcal{MAX} - \mathcal{MIN}$ Ant System. *Journal of Future Generation Computer Systems*, 16 :889–914, 2000.

[12] E.P.K. Tsang. *Foundations of Constraint Satisfaction*. Academic Press, London, UK, 1993.

[13] Marc van Dongen, Christophe Lecoutre, and Olivier Roussel. Results of the second csp solver competition. In *Second International CSP Solver Competition*, pages 1–10, 2007.

Une approche de contrôle autonome pour la recherche locale

Julien Robet Frédéric Lardeux Frédéric Saubion

LERIA, Université d'Angers
{nom}@info.univ-angers.fr

Résumé

Les algorithmes de recherche locale s'appuient principalement sur la notion de voisinage pour visiter des configurations successives au sein de l'espace de recherche, guidés également par les possibles stratégies de choix de ces voisins. Ces mouvements de proche en proche doivent être soigneusement contrôlés si l'on souhaite garantir une exploration suffisante de l'espace de recherche, ainsi que la possibilité de converger vers une solution. Des alternatives aux approches classiques ont été proposées afin d'envisager des voisinages plus complexes ou même de gérer simultanément plusieurs voisinages. Toutefois, biais souvent incontournable des métaheuristiques, ces approches induisent de nouveaux paramètres ou des heuristiques ad-hoc, dont l'importance sur les performances est grande mais le réglage délicat. Nous proposons ici, une approche de contrôle autonome pour un algorithme de recherche locale qui disposera de plusieurs opérateurs de mouvements, dont l'efficacité peut être diverse, et dont l'application respective sera ajustée en fonction de l'observation de l'état courant de la recherche, dans le but de s'adapter aux exigences d'équilibrage entre exploitation et exploration de l'espace des configurations. Nous définissons pour cela une nouvelle mesure de diversité, couplée à une mesure classique de qualité, qui autorise ainsi un contrôle simple de l'algorithme. Notre approche est finalement évaluée sur le problème classique d'affectation quadratique (QAP), obtenant des résultats prometteurs.

Abstract

Local search algorithms rely on the concept of neighborhood in order to visit successive configurations of the search space, also guided by the possible strategies for choosing these neighbors. These step by step moves must be carefully controlled if one wants to ensure a sufficient exploration of the serach space and the ability to converge towards a solution. Alternatives to conventional approaches have been proposed to consider neighborhoods or even to manage multiple neighborhoods. However, as an unavoidable bias of metaheuristics, these approaches induce new parameters or ad-hoc heuristics, whose impact on the performance is great but whose setting can be difficult. We propose here an approach for autonomous control of a local search algorithm, which has several moves operators, whose effeciency can be diverse and whose respective application is adjusted according to the observation of the current status of search, in order to adapt to the balance between exploitation and exploration of search space. To this aim, we define a new measure of diversification, coupled with a standard measure of quality, that allows us to easily control the algorithm. Our approach is finally experimented on the classical quadratic assignment problem (QAP), obtaining promising results.

1 Introduction

Les algorithmes de recherche locale [1, 23] sont des métaheuristiques qui ont été largement utilisées pour la résolution de problèmes combinatoires complexes aussi bien dans la communauté scientifique de la programmation par contraintes que dans celle de la recherche opérationnelle. Considérant un problème d'optimisation sous contraintes, ces méthodes de résolution s'appuient sur la définition d'une structure de voisinage au sein de l'espace de recherche, c'est à dire de l'ensemble des configurations possibles du problème. Une fois muni de cette relation de voisinage, l'algorithme explore alors cet espace en se déplaçant de proche en proche, guidé également par une fonction objectif qui permet d'évaluer la qualité des configurations rencontrées. Ces approches étant, par nature, incomplètes, elles permettent d'obtenir une solution sous optimale de bonne qualité, ou éventuellement une solution optimale. L'efficacité d'une telle procédure réside alors dans sa capacité à correctement explorer des zones variées de l'espace de recherche mais également dans sa propension à converger vers un optimum local,

relativement à la notion de voisinage, qui peut s'avérer, comme l'espère l'utilisateur, être également global. Le concept, particulièrement connu dans la communauté des algorithmes évolutionnaires, de balance entre intensification et diversification est un élément crucial à prendre en compte lors de la conception d'un algorithme de recherche locale. En effet, un des écueils classiques que rencontrent ces algorithmes est l'attraction excessive des minima locaux qui piègent le processus de recherche, lorsque l'ensemble des voisins potentiels sont de moins bonne qualité que la configuration courante, et que les stratégies de déplacement sont principalement fondées sur l'amélioration. Pour palier cet excès d'exploitation de l'espace de recherche (i.e., d'intensification) des mécanismes d'échappement ont alors été proposés dans un souci de diversification.

Les premières idées ont consisté à inclure, dans le choix du prochain mouvement à effectuer, des perturbations aléatoires [21], ou plus contrôlées comme dans le cas du recuit simulé [12]. Par la suite, une autre idée à consisté à utiliser d'autres fonctions (relations) de voisinages afin, par exemple, d'explorer plus largement l'espace par des voisinages de grande taille [2] ou encore, en utilisant plusieurs voisinages au sein d'un même algorithme [11, 19]. Si de telles approches permettent d'améliorer les performances des algorithmes, elles induisent par contre de nouveaux paramètres et/ou heuristiques qu'il convient de gérer correctement, ce qui constitue une tâche fastidieuse pour l'utilisateur. En effet, le réglage des paramètres reste un problème épineux pour le concepteur et l'utilisateur de telles métaheuristiques. Si l'ultime but de cette quête est de trouver des réglages universels permettant de traiter de larges classes de problèmes, l'utilisateur recourt, la plupart du temps, à une série d'expériences ainsi qu'à sa connaissance du problème pour tenter de formuler des règles empiriques d'ajustement de ces paramètres. On peut citer ici l'étude proposée par B. Mazure et al. [18], dans laquelle une valeur optimale de longueur de liste taboue est proposée pour une classe d'instances du problème SAT. Toutefois, il semble, de nos jours, préférable de s'orienter vers des algorithmes plus autonomes, incluant des mécanismes leur permettant de prendre en charge eux-même le réglage de leurs paramètres et de leurs heuristiques. Une telle approche nécessite alors l'intégration d'outils et techniques issues d'autres domaines de l'informatique, notamment de l'apprentissage artificiel (nous renvoyons le lecteur à [4, 10] pour plus de détails). Ce type d'approches de contrôle peut être schématisé par la figure 1.

Dans ce contexte, et inspirés par les travaux précédents que nous avons menés sur la gestion autonomes d'opérateurs multiples dans les algorithmes génétiques [15, 17], nous proposons ici une nouvelle approche pour concevoir un algorithme de recherche locale incluant plusieurs opérateurs de mouvements, correspondant à des voisinages et des stratégies de déplacement différents, dont le contrôle sera pris en charge automatiquement. Notre but est de montrer que nous pouvons bénéficier des atouts respectifs de ces opérateurs et décider de leur utilisation en fonction de la situation courante de la recherche. Pour cela, et afin d'ajuster convenablement la balance entre intensification et diversification, nous définissons un nouveau critère d'évaluation de la diversité pour la recherche locale qui sera couplé au critère, plus classique, de qualité, lié à la fonction objectif et aux contraintes du problème à résoudre. Nous utilisons ensuite ces mesures pour décider du prochain mouvement à effectuer. Afin d'évaluer les bénéfices de notre approche, nous avons expérimenté notre algorithme sur le fameux problème de l'affectation quadratique (QAP) qui a été largement étudié et pour lequel nous disposons d'une imposante libraire d'instances et de résultats [5].

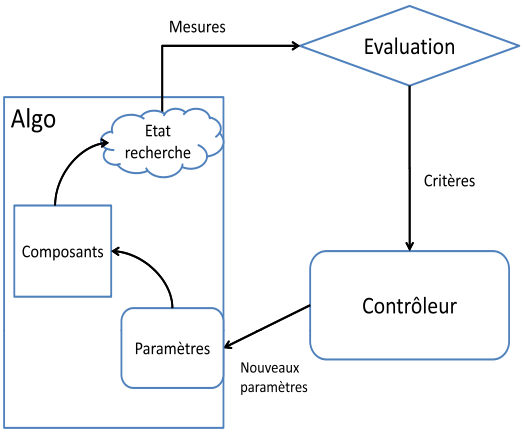


FIG. 1 – Schéma général

Dans la section 2 nous effectuons quelques rappels sur les principes de base de la recherche locale ainsi que sur les méthodes destinées à rendre la recherche locale plus autonome et adaptative. Dans la section 3, nous proposons notre nouvelle mesure de diversité pour l'étude des performances de la recherche locale. La section 4 présente le problème d'affectation quadratique sur lequel nous avons expérimenté notre approche ainsi que les opérateurs de mouvements que nous avons utilisés pour notre recherche locale. La section 5 présente notre algorithme ALS et enfin, la section 6 est consacrée à l'expérimentation de l'approche.

2 Vers une recherche locale autonome

Nous nous plaçons ici dans le contexte de la résolution de problèmes d'optimisation combinatoires classiquement définis par un ensemble de variables $Var = \{x_1, \cdots, x_n\}$, auxquelles sont associés des domaines de valeurs possibles $D = \{D_1, \cdots, D_n\}$, D_i étant l'ensemble des valeurs que peut prendre la variable x_i. L'espace de recherche S correspond au produit cartésien des domaines, $S = \Pi_{i=1}^{n} D_i$. Nous considérons un ensemble de contraintes C et une solution réalisable est alors une affectation de valeurs aux variables satisfaisant toutes les contraintes de C et respectant les domaines initiaux de D. Nous considérons également une fonction objectif $f : S \rightarrow \mathbb{R}$. Une solution optimale est une solution réalisable maximisant ou minimisant, selon le cas, la fonction f.

En programmation par contraintes, diverses approches sont envisageables pour résoudre de tels problèmes. On distingue classiquement les approches complètes, basées sur des parcours arborescents de l'espace de recherche couplés à des techniques d'élagage, et les approches incomplètes, principalement basées sur des méthodes métaheuristiques. L'exploration systématique de l'espace de recherche étant exclue, les métaheuristiques proposent des schémas généraux permettant de restreindre cette exploration à certaines zones. D'une manière très générale, deux stratégies fondamentales d'exploration et d'exploitation peuvent être identifiées :

- l'intensification, dont le but est de chercher une solution dans une zone réduite de l'espace,
- la diversification, qui permet de se déplacer d'une zone vers une autre zone, éventuellement lointaine, et autorise ainsi un balayage large de l'espace de recherche.

Ces notions sont mises en œuvre au sein des différentes métaheuristiques et l'efficacité relative d'une méthode réside essentiellement dans la gestion de leur complémentarité. La coordination de ces stratégies vise à éviter les problèmes induits par la présence d'optima locaux propres à la structure du problème traité et inhérents à ce type d'approche. Par exemple, dans le cas d'un algorithme de recherche par voisinage, une stratégie d'amélioration systématique de la configuration courante conduira le processus de recherche vers un optimum local où il risque de stagner.

Nous nous concentrons ici sur les méthodes de recherche locale (dites aussi par voisinage) [1, 23]. L'exploration se fait parmi les points voisins du point courant en s'appuyant sur une fonction qui estime leur qualité potentielle. Étant donné l'espace de recherche S, on se donne une fonction de voisinage $\mathcal{N} : S \rightarrow 2^S$ et une fonction d'évaluation $eval : S \rightarrow \mathbb{R}$, qui prend en compte la satisfaction des contraintes C du problème ainsi que la fonction objectif f. On définit deux actions élémentaires :

- l'amélioration qui consiste, à partir d'une configuration c, à choisir une configuration voisine c' dont la valeur $eval(c')$ améliore $eval(c)$ (cette notion varie selon que l'on veut maximiser ou minimiser le critère d'évaluation).
- le déplacement arbitraire qui consiste à choisir n'importe quelle configuration voisine c'.

L'efficacité de la recherche locale est alors liée à la gestion de la balance entre intensification et diversification. Cette gestion peut s'opérer, au sein de l'algorithme de recherche locale, par le biais de paramètres comportementaux, qui permettent d'ajuster l'utilisation des composants de l'algorithme (par exemple, la probabilité d'effectuer un mouvement aléatoire ou bien la longueur de la liste taboue), ou de paramètres structurels, qui sont directement liés aux composants eux-mêmes de l'algorithme (par exemple, le choix d'une fonction de voisinage, la fonction d'évaluation ou bien le choix d'une stratégie d'amélioration pour un processus de descente). Nous allons donc brièvement aborder ces deux aspects dans l'optique de la recherche locale autonome et nous recommandons au lecteur le récent ouvrage de R. Battiti et al. [4], pour plus de précisions.

2.1 Réglage de paramètres pour la recherche locale

Comme pour tout algorithme de type métaheuristique, le réglage des paramètres reste l'une des contraintes essentielles de leur utilisation pratique. Le réglage de paramètres a largement été étudié dans la communauté des algorithmes évolutionnaires [14]. Comme mentionné en introduction, cette tâche s'avère particulièrement ardue et repose sur le savoir-faire, bien souvent empirique, de l'utilisateur et sur ses connaissances des caractéristiques du problème à résoudre. Dès lors, l'ajustement de paramètres s'appuie sur une série, en général coûteuse, d'expériences. Une telle approche réduit considérablement l'universalité des valeurs obtenues et la transposition de ces expérimentations à d'autres problèmes. Une autre voie consiste alors à envisager un contrôle des paramètres durant l'exécution de l'algorithme. Ce contrôle peut être supervisé par un ensemble de connaissances acquises au préalable ou encore être abordé dans une optique plus autonome, les paramètres évoluant en fonction de l'état courant de la recherche. Le contrôle autonome de paramètres pour la recherche locale est largement illustré dans l'approche de recherche réactive [4]. De telles approches se développent à la frontière d'autres domaines de l'informatique, faisant notamment appel à des techniques issues de l'apprentissage

automatique ou de la programmation génétique.

En dehors des paramètres classiques et du point de vue des composants de l'algorithme, on peut également s'intéresser à faire évoluer la fonction d'évaluation en fonction de l'état de la recherche avec des systèmes de pénalités (voir également [4]) ou encore apprendre des stratégie adéquates de manière automatique [9]. Ici nous nous concentrons sur les aspects liés à la notion de voisinage.

2.2 Voisinages larges et multiples

Comme mentionné précédemment, le principe fondateur de la recherche locale est de parcourir l'espace de recherche selon une relation de voisinage définie entre les configurations possibles du problème. Le voisinage détermine ainsi la notion de structure de l'espace de recherche, on parle de paysage de recherche, dont l'importance est cruciale vis à vis des notions d'optima locaux ou globaux, évoquées plus haut. Dès lors, il parait naturel de s'intéresser à ce voisinage comme un moyen de gérer la balance entre intensification et diversification. On peut par exemple, pour diversifier la recherche, envisager d'utiliser des voisinages de tailles différentes, permettant de varier les possibilités de mouvements [11]. Il est également possible d'envisager des voisinages très larges [2]. Bien évidemment, l'extension du voisinage a un impact important sur le temps nécessaire à son évaluation pour décider du prochain mouvement.

2.3 Vers une approche englobante

Dans cet article, notre idée est de combiner finalement ces deux aspects (paramètres et composants) en considérant plutôt des opérateurs de mouvements en réglant automatiquement leur application au cours de la recherche. Nous introduisons donc, au sein de la recherche locale, une méthode de sélection d'opérateur adaptative, telle que nous l'avons étudiée dans [15] pour les algorithmes génétiques. L'objectif est donc d'évaluer, après application, l'impact des opérateurs et de se servir de cette évaluation pour ajuster leur utilisation au cours de la recherche. Cette approche peut être résumée par la figure 2.

Cette figure nous permet de mettre en lumière les principales questions liées au contrôle d'un algorithme de recherche locale :

- Comment évaluer la situation en cours ?
- Comment attribuer des récompenses aux opérateurs en fonction de cette évaluation ?
- Comment utiliser ces récompenses pour sélectionner l'opérateur à appliquer pour faire le prochain pas de recherche locale ?

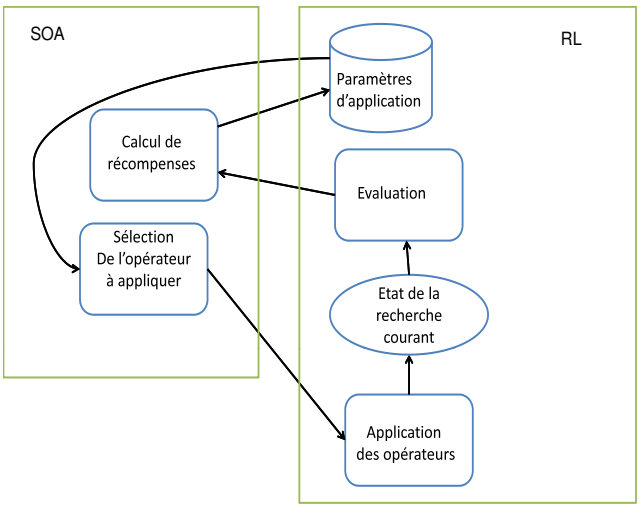


FIG. 2 – Recherche locale autonome

Alors que nous pouvons nous inspirer des mécanismes existants pour les algorithmes évolutionnaires [15] en ce qui concerne les deux derniers points, la question de l'évaluation est plus délicate. En effet, alors que dans le cas des algorithmes évolutionnaires, les critères de qualité et d'entropie de la population ont été étudiés [16, 17], il n'en est pas de même pour la recherche locale. En effet, si la notion de qualité permet également de guider et d'évaluer la recherche en cours, la notion de diversité mérite plus d'attention. Pourtant, il est apparu, dans nos travaux précédents [16, 17], que cette notion est très largement complémentaire de la qualité si l'on veut correctement gérer la balance entre diversification et intensification. Dans la section suivante, nos allons donc nous pencher sur la définition d'une nouvelle mesure de diversité adaptée à la recherche locale.

3 Une mesure de diversité pour la recherche locale

Le concept de diversité est souvent employé pour caractériser un parcours de recherche locale. Il est assez intuitif et pourrait se résumer ainsi : un parcours est d'autant plus diversifié qu'il a visité des régions éloignées de l'espace de recherche. La diversification est un élément de première importance pour un système de résolution dans le sens où elle évite de focaliser la fouille de l'espace de recherche dans des zones confinées, et tend à orienter le système vers des régions attractives.

Cette notion, bien que couramment évoquée par les concepteurs de métaheuristiques, est difficilement calculable en pratique. En effet, une telle mesure dépend de plusieurs éléments comme, par exemple, la façon dont est codée une configuration, la taille de l'espace de recherche. Le cas des méthodes de type évolutionnaire doit cependant être traité séparément. En effet, un algorithme manipulant une population de configurations peut évaluer la diversité de la recherche en comparant les différents individus composant la population à chaque itération. En général, ce procédé consiste à évaluer une distance moyenne entre les individus, par exemple à l'aide d'un calcul d'entropie. Le lecteur intéressé trouvera des informations à ce sujet dans [6].

Les méthodes à base de recherche locale pure ne peuvent pas bénéficier de ces techniques, du fait qu'elles ne traitent qu'une seule configuration à chaque pas de la recherche. La diversité de cette dernière doit donc être calculée en fonction du parcours effectué.

3.1 Pré-requis pour une mesure pertinente

Malgré le succès et l'étude poussée des méthodes de recherche locale durant ces dernières années, peu de travaux ont été menés dans le but de quantifier la diversité de tels procédés. A. Linhares introduit dans [13] les notions relatives au sujet, définit les caractéristiques que doit respecter une telle mesure, et propose une méthode d'évaluation de la diversité basée sur les mesures de distance entre configurations. Nous nous efforçons donc de respecter les contraintes suivantes pour notre système d'évaluation de diversité qui se présentera comme une fonction div :

- son domaine sera le parcours P assimilé à l'ensemble des configurations qui le composent.
- son co-domaine sera l'intervalle fermé $[0, 1]$. En effet, cette normalisation permet d'obtenir des valeurs précises, indépendantes de la taille du problème et de celle du parcours.
- $div(P) = 0$ ssi $|P| = 0$

3.2 Vers une nouvelle mesure

Afin de quantifier la diversité d'un parcours de recherche locale, la mesure que nous proposons consiste à analyser, pour chaque variable décisionnelle, la répartition des valeurs affectées dans son domaine respectif. Notre approche part en effet d'un constat simple : dans le cas où un parcours de recherche est très diversifié, les valeurs successives affectées aux différentes variables auront tendance à "couvrir" leur domaine. Dans le cas contraire, un parcours peu diversifié sera caractérisé par l'affectation d'un faible nombre de valeurs différentes à chaque variable (relativement au cardinal de son domaine), spécialement pour celles dont la valeur reste inchangée pendant le parcours étudié. L'évaluation consiste donc à observer, pour chaque variable décisionnelle, l'écart type du nombre d'occurrences de chaque valeur possible pour cette variable (i.e., pour chaque élément de son domaine). La moyenne de ces écarts type traduit la similitude du parcours étudié. En effet, les écarts types calculés seront d'autant plus faibles que le parcours étudié sera diversifié. Afin d'être normalisée, la mesure est ensuite divisée par la similitude maximale théoriquement atteignable. On obtient donc une valeur comprise entre 0 et 1 que l'on va retrancher de 1 pour traduire la diversité du parcours.

Plus formellement, étant donné un espace de recherche $S, P \subseteq S$ est un parcours de recherche locale constitué d'un ensemble de $|P|$ configurations. Soit $x \in Var$ une des variables décisionnelles du problème traité. On définit alors $aff(x, c)$ la valeur que prend la variable x pour la configuration $c \in P$ ($aff(x, c) \in D_x$).

Nous nous intéressons au nombre d'affectations donnant la valeur v à la variable x au cours du parcours P, que nous définissons comme :

$$nbutil(x, v, P) = |\{c \in P | aff(x, c) = v\}|$$

L'écart type du nombre d'occurrences des différentes valeurs prises par la variable x est donné par :

$$et(x, P) = \sqrt{\frac{\Sigma_{v \in D_x} \left(nbutil(x, v, P) - (\frac{|P|}{|D_x|})\right)^2}{|D_x|}}$$

L'écart type maximal atteignable lors d'un parcours P pour la variable x (cas où une seule valeur est utilisée pour chaque configuration de P) est alors :

$$etmax(x, P) = \sqrt{\frac{(|P| - 1).(\frac{|P|}{|D_x|})^2 + (|P| - \frac{|P|}{|D_x|})^2}{|D_x|}}$$

La similitude des configurations qui composent le parcours P étudié peut donc maintenant être définie par :

$$sim(P) = \frac{\Sigma_{x \in Var} et(x, P)}{|Var|}$$

La similitude maximale que l'on peut obtenir avec un parcours P (où toutes les configurations sont identiques) est :

$$simmax(P) = \frac{\Sigma_{x \in Var} etmax(x, P)}{|Var|}$$

La diversité est alors définie comme :

$$div(P) = \begin{cases} 0 \text{ si } |P| = 0 \\ 1 - \frac{sim(P)}{simmax(|P|)} \text{ sinon} \end{cases}$$

Deux cas extrême peuvent être identifiés :

- Un parcours P_1 a une diversité minimale si toutes les configurations de P_1 sont égales, c'est-à-dire que chaque variable décisionnelle x prend une valeur $|P_1|$ fois, et que les $(|D_x| - 1)$ autres valeurs ne sont jamais utilisées.
- Un parcours P_2 a une diversité maximale ssi $\forall x \in Var, \nexists (v_1, v_2) \in D_x^2, nbutil(x, v_1, P_2) - nbutil(x, v_2, P_2) > 1$. Un tel niveau n'est pour ainsi dire jamais atteint en pratique. On peut par exemple l'obtenir en affectant à chaque variable la valeur de son domaine qu'elle a le moins utilisée, et ceci à chaque itération de la recherche.

4 Application au problème d'affectation quadratique

4.1 Présentation du problème et travaux existants

Nous avons choisi de valider notre approche sur le célèbre problème de l'affectation quadratique (quadratic assignment problem, QAP), qui a été largement étudié au cours des cinquante dernières années. Intuitivement, le problème consiste à affecter des ressources à différents sites, en prenant en compte les distances qui séparent ces sites ainsi que l'importance des flux de données échangés entre les différentes ressources. Plus formellement, étant donnés n ressources et n sites, on dispose de deux matrices de taille $n \times n$, $D = [d_{i,j}]$ et $F = [f_{k,l}]$, où $d_{i,j}$ la distance entre les sites i et j et $f_{k,l}$ est la quantité d'informations transitant entre les ressources k et l. On va alors chercher une affectation π (i.e., une permutation de $1..n$) des ressources aux sites de coût $f(\pi)$ minimal. Le coût d'une telle affectation étant calculé comme suit :

$$f(\pi) = \Sigma_{i=1}^{n} \Sigma_{j=1}^{n} d_{i,j} \cdot f_{\pi(i),\pi(j)}$$

Le problème a été démontré NP-difficile [20], et est parfois qualifié du plus difficile de cette classe de complexité, du fait que de nombreux problèmes, tels que ceux du voyageur de commerce, de la clique maximale ou de partitionnement de graphe, sont un cas particulier d'affectation quadratique. D'autre part, il présente une forte importance pratique due à sa capacité à modéliser de nombreuses situations issues du domaine industriel. Cependant, malgré la quantité de travaux qu'a suscité le problème, même les méthodes les plus performantes ne peuvent résoudre à l'optimum des instances de taille supérieur à 30. Parmi ces dernières, on citera notamment deux versions de recherche taboue. La première, nommée Robust taboo search [22], a été proposée par Taillard en 1991 qui reste à ce jour une des plus compétitive. La seconde, qui vise à régler la taille de la liste taboue automatiquement via une détection de cycles performante, a été mise au point par Battiti et Tecchiolli en 1994 sous le nom de Reactive tabu search [3]. Fleurent et Ferland ont également obtenu d'excellents résultats avec un algorithme de type génétique [8]. Enfin, notons que la méthode de recuit simulé ne semble pas vraiment adaptée au problème puisque la meilleure version de ce type d'algorithme appliquée au QAP, présentée par Connolly en 1990 [7], ne permet pas de rivaliser avec les approches évoquées plus haut. Notons par ailleurs que les résultats de chacune d'elle dépendent fortement du type d'instance sur laquelle elle est appliquée, de telle sorte qu'il est inapproprié de distinguer une d'entre elles comme supérieure aux autres. Enfin, signalons que le QAP dispose d'une librairie fournie et variée d'instances de problèmes [5] et de résultats, permettant d'évaluer les apports de notre mécanisme. Pour une étude plus approfondie du problème, nous renvoyons le lecteur intéressé au site `http ://www.seas.upenn.edu/qaplib/`.

En ce qui concerne les approches de résolution par recherche locale, il est évident que, étant donnée une permutation π correspondant à une affectation de ressources aux sites, un voisinage basique consiste à échanger deux ressources, puisque l'on souhaite respecter la contraintes d'affectation bijective et retrouver une permutation. Ainsi, on peut ensuite définir des voisinages plus larges autorisant plus d'échanges entre les sites. Nous allons donc nous servir de ces voisinages pour définir nos opérateurs de mouvements.

4.2 Opérateurs candidats

Rappelons ici que le but du présent travail n'est pas d'obtenir des résultats concurrentiels pour le QAP mais de mettre en place un contrôleur capable de tirer le meilleur profit d'un ensemble d'opérateurs aux caractéristiques a priori inconnues. Il est donc intéressant de proposer au contrôleur des opérateurs volontairement variés pour que le système puisse mettre en avant une éventuelle complémentarité. Ainsi, nous avons défini un jeu de dix opérateurs, dont les cinq premiers paraissent naturellement voués à l'intensification et cinq autres présentant une tendance à la diversification.

O1 consiste à échanger deux ressources de telle sorte que chacune se retrouve placée sur le site précédemment occupé par l'autre. L'ensemble des paires candidates à un tel échange (qui sont au nombre de $\frac{n.(n-1)}{2}$) sont examinées et la meilleure

(en terme de gain pour la fonction d'évaluation) est sélectionnée.

O2 est similaire à O1, mais ne nécessite pas systématiquement l'exploration de l'ensemble du voisinage car la première paire améliorante (qui présente un gain strictement positif) est choisie.

O3 fonctionne comme O1, mais n'opte pas obligatoirement pour la meilleure paire. A la place, il en choisit aléatoirement une parmi le sous-ensemble des k meilleures paires rencontrées. Nous avons fixé $k = 5$ pour nos expérimentations.

O4 reprend le principe de O1, mais avec une profondeur de 2 au lieu de 1, c'est-à-dire que les deux meilleures paires indépendantes p_1 et p_2 sont échangées simultanément. Par indépendantes, on entend qu'une unité ne peut appartenir à p_1 et p_2 à la fois.

O5 est identique à O4 mais avec une profondeur de 3.

O6 effectue des échanges entre 3 ressources. L'échange est obligatoirement complet, c'est-à-dire que les trois ressources impliquées dans l'échange doivent être déplacées. On a donc ici deux choix possibles, parmi lesquels celui apportant le meilleur gain est choisi. Les 3 ressources impliquées sont sélectionnées aléatoirement.

O7 est identique à O6, mais avec des échanges entre 4 ressources.

O8 est identique à O6, mais avec des échanges entre 5 ressources.

O9 est identique à O6, mais avec des échanges entre 6 ressources.

O10 effectue une succession de k' (fixé à 3 pour les expérimentations) échanges de 2 ressources choisies aléatoirement.

5 L'algorithme ALS

Le but de notre système, nommé ALS (Autonomous Local Search) est de gérer un ensemble d'opérateurs de recherche locale afin de les appliquer au moment où ils peuvent le plus améliorer la recherche. Le défit de ce procédé est donc de faire cohabiter les trois principaux modules que sont l'évaluation de l'état courant de la recherche, l'attribution de récompenses aux composants internes, et l'utilisation de ces récompenses pour choisir l'opérateur à appliquer.

5.1 Évaluation des opérateurs

Le procédé utilisé pour analyser les composants internes est largement inspiré de nos précédents travaux dans le domaine de la recherche évolutionnaire [16]. Son principe est de maintenir tout au long de la recherche un historique des performances récentes de chaque opérateur. L'originalité de la méthode vient du fait que son analyse ne se contente pas d'un seul critère de jugement (les variations de qualité apportées par un opérateur), mais examine également les écarts en diversité. C'est ici que la mesure de diversité que nous proposons section 3.2 prend toute son importance. Formellement, étant donné un opérateur on définit :

- $\Delta Q_{op,t}$ la variation moyenne de qualité résultant des t applications de op
- $\Delta D_{op,t}$ la variation moyenne de diversité résultant des t applications de op

Le paramètre t correspond à la taille de la fenêtre glissante qui stocke les informations relatives à chaque opérateur. Étant donné que nous cherchons à faire ressortir les caractéristiques des composants à un moment précis de la recherche, il est important de régler ce paramètre correctement. S'il prend une valeur trop élevée, alors l'évaluation risque de ne pas refléter les spécificités actuelles. À l'inverse, une valeur trop faible ne permettra pas de mémoriser les cas où un opérateur devient très efficace sur une courte période. La variation de qualité apportée par l'application de op, traditionnellement définie par $\Delta Q = eval(op(c)) - eval(c)$ a été redéfinie afin de prendre en compte le fait qu'une configuration médiocre est plus facilement améliorable qu'une autre de bonne qualité. On a alors, après pondération :

$$\Delta Q = \frac{eval(op(c)) - eval(c)}{eval(c) + 1}$$

Enfin, les écarts de diversité entre deux itérations sont calculés ainsi :

$$\Delta D = div(P_{i,j}) - div(P_{i-1,j-1})$$

où $P_{i,j}$ est l'ensemble des configurations visitées de l'itération i à l'itération j.

À partir de cela, nous mettons en place le système de récompenses suivant :

$$score(op) = \alpha.\Delta Q_{op,t} + (1 - \alpha).\Delta D_{op,t}$$

où $\alpha \in [0..1]$ et désigne l'orientation que doit adopter la recherche. Une forte (respectivement faible) valeur favorise l'intensification (respectivement la diversification). Les différentes probabilités d'application sont ensuite déduites de ces récompenses :

$$p(op_k) = p_{min} + max\Big(0, (1 - p_{min}).\frac{score(op_k)}{\sum_{i=1}^{nbop} score(op_i)}\Big)$$

La probabilité $1 \leq p_{min} \leq \frac{1}{nbOp}$ ($nbop$ représentant le nombre total d'opérateurs) assure à l'ensemble des opérateurs une chance d'être élu à chaque pas de la recherche dans le but de garantir une diversité minimale dans les sélections effectuées.

5.2 Évaluation de la recherche

En appliquant les mêmes principes de mémorisation aux fluctuations de la recherche, on obtient de précieuses informations quant à l'évolution dans l'espace de recherche. En particulier, il est utile de s'intéresser aux écarts de diversité apportés par les dernières itérations. Une perte en diversité traduit la fouille d'une zone précise de l'espace de recherche, alors qu'un gain apparaît lors de l'éloignement d'une certaine région.

5.3 Stratégies d'application

Au cours de la résolution, le choix de l'opérateur à appliquer est effectué selon les probabilités évoquées dans la section 5.1. Comme nous l'avons vu précédemment, ces probabilités sont largement influencées par un paramètre α qui modélise la balance souhaitée entre intensification et diversification. La stratégie d'application peut donc être vue comme le moyen de calculer la valeur de α en fonction des observations collectées sur l'état de la recherche. Par exemple, si l'on détecte un blocage dans un optima local, il est bénéfique de faire décroître α afin de favoriser la diversification du parcours. Ainsi, à l'heure actuelle, nous avons doté ALS d'une stratégie simpliste ($alpha \in \{0, 1\}$ au lieu de $\alpha \in [0..1]$) afin, dans un premier temps, de vérifier l'impact de la méthode : il s'agit de définir la valeur par défaut de α à 1 (intensification pure) et de ne procéder à une diversification que lorsque l'on en détecte le besoin. Ce besoin est détecté par une perte de diversité, couplée à une absence de gains en qualité, à la suite de l'application d'un opérateur. On fixe alors α à 0 pour l'itération suivante, dans le but d'effectuer un pas de diversification.

6 Résultats expérimentaux

Nous avons expérimenté ALS sur 38 instances issues de la QAPLIB [5]. Une première variante, disposant des 10 opérateurs présentés dans la section 4.2, et une seconde ne manipulant que 2 opérateurs (nommés O1 et O9 dans la section 4.2, respectivement meilleurs intensificateur et diversificateur lors des tests avec ALS-10) ont été comparées à un choix uniforme parmi les 10 opérateurs. Le tableau 1 représente les résultats obtenus par les 3 variantes, et indique à titre indicatif ceux de l'algorithme Robust Taboo Search, présenté en section 4.1. Pour chaque instance, la meilleure valeur connue (mars 2009) pour la fonction objectif est indiquée dans la colonne BKV (Best Known Value). Les résultats indiqués représentent la déviation moyenne θ_{avg} par rapport à BKV qui est calculée selon la formule :

$$\theta_{avg} = \frac{100(f_{avg} - BKV)}{BKV}$$

où f_{avg} représente la valeur moyenne de la fonction objectif sur 10 exécutions, chacune d'elles ayant duré un nombre de secondes spécifié dans la colonne temps.

Les résultats du tableau 1 font clairement ressortir l'intérêt du contrôleur puisque sur les 38 instances testées, seulement six ont apporté de meilleurs résultats avec un choix uniforme. De plus, cinq d'entre elles appartiennent à la famille bur26 où ALS obtient des résultats très semblables. Pour les instances structurées, comme sko* ou tai*a, notre approche semble avoir un très bon comportement (le choix uniforme n'obtient qu'une seule fois de meilleurs résultats). Notons également que l'élargissement de la gamme d'opérateurs manipulés par le système est bénéfique pour l'efficacité de la résolution. En effet, ALS-10 a été plus performant qu'ALS-2 pour 31 des 38 instances testées.

7 Conclusion

Ce travail pose les bases du socle sur lequel nous comptons nous appuyer pour développer diverses stratégies, et mener nombre d'expérimentations. En effet, les différents composants impliqués nécessitent une étude plus poussée afin d'accroître la qualité de leur contribution au système. Notamment, le module chargé de maintenir les besoins en intensification et en diversification, un élément crucial de la démarche, est loin d'avoir atteint un comportement optimal. Des recherches complémentaires à ce sujet devraient significativement améliorer l'approche. Une seconde piste intéressante réside dans l'élargissement de la gamme des opérateurs candidats que manipule le contrôleur. A l'heure actuelle, les dix opérateurs utilisés sont plutôt simples et ont été définis *à la main*. L'étape suivante consistera à définir de nouveaux opérateurs reproduisant, sur un nombre limité d'itérations, le comportement des méthodes les plus efficaces pour le problème étudié (en l'occurrence celui de l'affectation quadratique). Nous prévoyons également de mettre en place un procédé visant à générer automatiquement de nouveaux opérateurs. Le contrôleur prendrait alors une fonction double puisqu'en plus d'organiser la séquence d'applications des différents opérateurs, il permettrait d'évaluer les caractéristiques de ceux régulièrement générés, et ainsi conserver les plus prometteurs d'entre eux.

Instance	BKV	temps	ALS 10 op	ALS 2 op	CU 10 op	Ro-TS
bur26a	5426670	50	0,0701	**0,0698**	0,1177	0,0004
bur26b	3817852	50	0,1244	0,1274	**0,1084**	0,0032
bur26c	5426795	50	**0,0236**	0,0266	0,0359	0,0004
bur26d	3821225	50	**0,0205**	0,0218	0,0214	0,0015
bur26e	5386879	50	0,0462	0,1260	**0,0366**	0
bur26f	3782044	50	0,1360	0,1019	**0,0153**	0,0007
bur26g	1,0E+07	50	0,1239	0,1621	**0,0251**	0,0003
bur26h	7098658	50	0,0622	0,2453	**0,0164**	0,0027
chr25a	3796	40	18,3878	**15,9484**	42,4341	6,9652
els19	1,7E+07	20	**1,3656**	7,4253	4,8532	0
kra30a	88900	76	**1,0146**	1,6097	3,8774	0,4702
kra30b	91420	86	**0,0831**	0,2111	2,4251	0,0591
tai20b	1,2E+08	27	**0,1399**	0,1810	1,6870	0
tai25b	3,4E+08	50	**0,1891**	4,2186	1,7558	0,0072
tai30b	6,4E+08	90	**0,8338**	4,1695	1,5794	0,0547
tai35b	2,8E+08	147	2,2365	2,4788	**1,4712**	0,1777
tai40b	6,4E+08	240	**1,5322**	3,7357	2,0287	0,2082
tai50b	4,6E+08	480	1,5832	**0,9594**	1,4307	0,2943
tai60b	6,1E+08	855	**0,5568**	0,7787	1,4344	0,3904
tai80b	8,2E+08	2073	**1,1055**	1,5086	1,5163	1,4354
nug20	2570	30	**0,2802**	0,5058	1,9767	0,101
nug30	6124	83	**0,2743**	0,3919	2,0901	0,271
sko42	15812	248	**0,1505**	0,2416	2,0529	0,187
sko49	23386	415	**0,3464**	0,3831	2,0174	0,198
sko56	34458	639	**0,2969**	0,3419	2,1139	0,347
sko64	48498	974	**0,3784**	0,4555	2,1989	0,221
sko72	66256	1415	**0,5162**	0,5687	2,4864	0,478
sko81	90998	2041	4,1135	**4,0523**	5,3981	0,304
sko90	115534	2825	0,4368	**0,4130**	2,2731	0,386
tai20a	703482	26	2,0103	**1,5700**	3,6558	0,769
tai25a	1167256	50	**1,8540**	1,8858	3,8223	1,128
tai30a	1818146	87	1,8407	**1,6269**	3,6971	0,871
tai35a	2422002	145	**1,6184**	1,8995	3,9348	1,356
tai40a	3139370	224	**1,6309**	2,0021	4,0596	1,284
tai50a	4941410	467	**1,6362**	2,0849	4,4437	1,377
tai60a	7208572	820	**1,5267**	2,1955	4,7171	1,544
tai80a	1,4E+07	2045	**1,0563**	2,3685	4,7724	1,170
wil50	48816	441	**0,0639**	0,1135	1,0005	0,137

TAB. 1 – Déviation moyenne d'ALS (avec 2 et 10 opérateurs) et d'un choix uniforme parmi les 10 opérateurs par rapport aux meilleurs résultats connus (BKV)

Références

[1] Emile Aarts and Jan Karel Lenstra, editors. *Local Search in Combinatorial Optimization*. Princeton University Press, 2003.

[2] Ravindra K. Ahuja, Krishna C. Jha, Krishna C. Jha, James B. Orlin, James B. Orlin, Dushyant Sharma, and Dushyant Sharma. A survey of very large-scale neighborhood search techniques. *Discrete Applied Mathematics*, 123 :75–102, 2002.

[3] R. Battiti and Tecchiolli. The reactive tabu search. *ORSA Journal on Computing*, 6(2) :126–140, 1994.

[4] Roberto Battiti, Mauro Brunato, and Franco Mascia. *Reactive Search and Intelligent Optimization*, volume 45 of *Operations research/Computer*

Science Interfaces. Springer Verlag, 2008.

[5] R. E. Burkard, S. Karisch, and F. Rendl. Qaplib-a quadratic assignment problem library. *European Journal of Operational Research*, 55(1) :115–119, November 1991.

[6] Edmund K. Burke, Steven M. Gustafson, Graham Kendall, and Natalio Krasnogor. Advanced population diversity measures in genetic programming. In *Parallel Problem Solving from Nature - PPSN VII, 7th International Conference*, volume 2439 of *Lecture Notes in Computer Science*, pages 341–350. Springer, 2002.

[7] David T. Connolly. An improved annealing scheme for the qap. *European Journal of Operational Research*, 46(1) :93–100, May 1990.

[8] Charles Fleurent, Jacques, and A. Ferland. Genetic hybrids for the quadratic assignment problem. In *DIMACS Series in Mathematics and Theoretical Computer Science*, pages 173–187. American Mathematical Society, 1994.

[9] A. Fukunaga. Automated discovery of local search heuristics for satisfiability testing. *Evolutionary Computation*, 16(1) :31–61, 2008.

[10] Youssef Hamadi, Eric Monfroy, and Frederic Saubion. Special issue on autonomous searcch. *Contraint Programming Letters*, 4, 2008.

[11] Pierre Hansen and Nenad Mladenovi. A tutorial on variable neighborhood search. Technical report, Les Cahiers du GERAD, HEC Montreal and GERAD, 2003.

[12] Scott Kirkpatrick, D. Gelatt Jr., and Mario P. Vecchi. Optimization by simmulated annealing. *Science*, 220(4598) :671–680, 1983.

[13] Alexandre Linhares. The structure of local search diversity. In *Math'04 : Proceedings of the 5th WSEAS International Conference on Applied Mathematics*, pages 1–5, Stevens Point, Wisconsin, USA, 2004. World Scientific and Engineering Academy and Society (WSEAS).

[14] Fernando G. Lobo, Cláudio F. Lima, and Zbigniew Michalewicz, editors. *Parameter Setting in Evolutionary Algorithms*, volume 54 of *Studies in Computational Intelligence*. Springer, 2007.

[15] J. Maturana, A. Fialho, F. Saubion, M. Schoenauer, and M. Sebag. Compass and dynamic multi-armed bandits for adaptive operator selection. In *Proceedings of IEEE Congress on Evolutionary Computation CEC*, 2009. to appear.

[16] J. Maturana and F. Saubion. Towards a generic control strategy for EAs : an adaptive fuzzy-learning approach. In *Proceedings of IEEE International Conference on Evolutionary Computation (CEC)*, pages 4546–4553, 2007.

[17] Jorge Maturana and Frederic Saubion. A compass to guide genetic algorithms. In G. Rudolph et al., editor, *Proc. PPSN'08*, pages 256–265. Springer, 2008.

[18] Bertrand Mazure, Lakhdar Sais, and Éric Grégoire. Tabu search for sat. In *AAAI/IAAI*, pages 281–285, 1997.

[19] Nenad Mladenovic and Pierre Hansen. Variable neighborhood search. *Computers & OR*, 24(11) :1097–1100, 1997.

[20] Sartaj Sahni and Teofilo F. Gonzalez. P-complete approximation problems. *J. ACM*, 23(3) :555–565, 1976.

[21] Bart Selman, Henry A. Kautz, and Bram Cohen. Noise strategies for improving local search. In *AAAI*, pages 337–343, 1994.

[22] Éric D. Taillard. Robust taboo search for the quadratic assignment problem. *Parallel Computing*, 17(4-5) :443–455, 1991.

[23] Pascal Van Hentenryck and Laurent Michel. *Constraint-Based Local Search*. The MIT Press, 2005.

Analyse de conflits dans le cadre de la recherche locale*

Gilles Audemard **Jean-Marie Lagniez** **Bertrand Mazure** **Lakhdar Saïs**

Université Lille-Nord de France, Artois, F-62307 Lens
CRIL, F-62307 Lens
CNRS, UMR 8188, F-62307 Lens
{audemard,lagniez,mazure,sais}@cril.fr

Résumé

Dans cet article, nous présentons une nouvelle approche pour sortir des minimums locaux dans le cadre de la recherche locale. Cette approche est basée sur le principe d'analyse de conflits utilisé dans les solveurs SAT modernes. Nous proposons une extension du graphe d'implications au cadre de la recherche locale où plusieurs conflits sont présents pour une interprétation donnée. Nous présentons ensuite une méthode basée sur la propagation unitaire, permettant de construire et d'exploiter de tels graphes. Enfin, nous étendons le schéma classique de WSAT pour y intégrer notre analyse de conflits. Les résultats expérimentaux montrent que l'intégration de notre système d'analyse de conflits améliore sensiblement les performances de WSAT sur les problèmes structurés. De plus, cette méthode isolant des sous-problèmes inconsistants, est capable de montrer que l'instance n'admet pas de modèle.

1 Introduction

La résolution pratique du problème SAT peut être divisée en deux catégories. D'un coté, les méthodes de recherche locale [16, 15, 10], incomplètes, et de l'autre les méthodes complètes, basées la plupart du temps sur la procédure de Davis et Putnam [3]. Les premières explorent l'espace de recherche de manière stochastique et ne savent, dans la plupart des cas, répondre uniquement lorsque la formule est satisfaisable. Autant en recherche opérationnelle elles ont montré leur efficacité, autant, dans le cas du problème SAT, les méthodes de recherche locale ne sont compétitives que pour les problèmes aléatoires.

*supporté par le projet ANR UNLOC

D'un autre coté, l'amélioration constante des méthodes complètes permet depuis quelques années la résolution de larges instances issues de problèmes industriels. L'un des composants clés de ces solveurs modernes est l'analyse de conflits [18]. Elle fait partie des contributions ayant le plus apportées, en terme d'efficacité, pour la résolution pratique de telles instances. Les solveurs actuels de type CDCL (*Conflict Driven, Clause Learning*), tel que Berkmin [8], Minisat [5] et d'autres, implantent une telle approche. Lors de l'analyse d'un conflit un niveau de backtrack est calculé, un « *nogood* » est extrait et ajouté à la base. Cette sauvegarde permet d'éviter de parcourir des sous-espaces insatisfaisables.

Dans cet article, nous proposons d'étendre l'analyse de conflits [18] dans le cadre des algorithmes de recherche locale stochastiques. L'objectif est double. Premièrement, nous souhaitons exploiter cette analyse et les nogoods résultants pour sortir des minimums locaux. En effet, la stratégie utilisée pour sortir des minimums locaux joue un grand rôle dans l'efficacité des méthodes de recherche locale. Il est à noter qu'une approche semblable à déjà été proposée dans [2] mais était limitée à des résolutions sur deux clauses. L'autre objectif est de répondre à un des challenges importants de la communauté scientifique : proposer une méthode de recherche locale qui puisse répondre à l'insatisfaisabilité d'une formule [17]. Cela est possible car les nogoods déduits par l'analyse des conflits vont être ajoutés à la formule initiale. Cette extension présente des difficultés inhérentes aux méthodes de recherche locales. Celles-ci utilisent des interprétations complètes dans lesquelles plusieurs clauses peuvent être falsifiées. L'absence de notion

de niveau de décision participe également à cette difficulté.

L'article est organisé de la façon suivante. Après quelques définitions préliminaires (section 2), nous présentons le schéma de base d'une recherche locale de type WSAT ainsi que le graphe d'implications classique et son utilisation (section 3). Dans la section 4, nous proposons dans un premier temps une façon d'étendre la notion de graphe d'implications à une interprétation complète. Ensuite, une méthode basée sur la propagation unitaire pour la construction de tels graphes est proposée. Pour terminer, l'intégration de cette analyse à un solveur de type WSAT est décrite. Avant de conclure, nous donnons des résultats expérimentaux dans la section 5.

2 Définitions et notations

Le problème SAT est un problème de décidabilité consistant à savoir si une formule propositionelle Σ mise sous forme CNF (*Conjontive Normal Form*) est logiquement valide. Une formule *CNF* Σ est un ensemble (interprété comme une conjonction) de *clauses*, où une clause est un ensemble (interprété comme une disjonction) de *littéraux*. Un littéral est soit positif (x) soit négatif ($\overline{x}$). Les deux littéraux x et $\overline{x}$ sont appelés littéraux *complémentaires*. L'ensemble des variables apparaissant dans Σ sera noté $\mathcal{V}_\Sigma$. Une interprétation $\mathcal{I}$ d'une formule booléenne Σ associe une valeur $\mathcal{I}(x)$ aux variables $x \in \mathcal{V}_\Sigma$. L'interprétation $\mathcal{I}$ est dite *complète* si elle associe une valeur de vérité à chaque $x \in \mathcal{V}_\Sigma$, sinon elle est dite *partielle*. Une interprétation peut également être représentée par un ensemble de littéraux. Un *modèle* d'une formule de Σ est une interprétation $\mathcal{I}$ qui satisfait la formule, ce qui est noté $\mathcal{I} \models \Sigma$. La négation d'une formule Γ sera notée $\overline{\Gamma}$. Les notations suivantes sont utilisées par la suite :

- $\Sigma_{|_x}$ dénote la formule Σ simplifiée par l'affectation du littéral x à la valeur vrai. Cette notation est étendue aux interprétations : soit $\mathcal{P} = \{x_1, ..., x_n\}$ une interprétation, on définit $\Sigma_{|_\mathcal{P}} = (...(\Sigma|_{x_1})...|_{x_n})$;
- Σ^* dénote la formule close Σ après application de la propagation unitaire ;
- $\models_*$ dénote la déduction logique par propagation unitaire : $\Sigma \models_* x$ signifie que le littéral x est déduit par propagation unitaire à partir de Σ. On écrit $\Sigma \models_* \bot$ si la formule est inconsistante (insatisfaisable) par propagation unitaire.

3 Préliminaires

3.1 Algorithmes de recherche locale

Le schéma de recherche locale est relativement simple. Celui-ci consiste à se déplacer stochastiquement dans l'ensemble des interprétations jusqu'à l'obtention d'une solution. À chaque étape, on essaie de réduire le nombre de clauses non satisfaites (c'est l'étape communément appelé descente). L'interprétation suivante est choisie parmi l'ensemble des interprétations voisines (c'est-à-dire différant uniquement sur la valeur d'une seule variable) de l'interprétation courante. Lorsqu'aucune descente n'est possible, on se trouve alors dans un minimum local. L'un des points cruciaux des méthodes de recherche locale est la technique utilisée pour s'échapper de ces minimums locaux. De nombreuses approches ont été proposées dans la littérature : liste tabou [13], recuit simulé, ajout des résolvantes [2]... D'autres améliorations (choix de la variable à flipper, propagation unitaire, ...) ont également été proposé, le lecteur intéressé pourra se référer à [14] pour une description plus détaillée. Nous présentons ci-dessous l'algorithme WSAT que nous utilisons comme base de la nouvelle approche que nous proposons dans ce papier.

Algorithme 1 : WSAT

```
Input : Σ une formule CNF
Output : SAT si un modèle de Σ est trouvé,
         UNKNOWN sinon
1  for i ← 1 to MaxTries do
2      I_c ← interprétation complète de Σ;
3      for j ← 1 to MaxFlips do
4          if I_c ⊨ Σ then
5              return SAT;
6          α ∈ Σ tel que I_c ⊭ α;
7          if ∃x ∈ α permettant une descente then
8              flipper(x)
9          else                    /* minimum local */
10             Remplacer I_c par une interprétation
               obtenu selon un critère d'échappement;
11 return UNKNOWN;
```

L'algorithme WSAT (algorithme 1) génère une interprétation complète et et la modifie jusqu'à obtenir un modèle ou jusqu'à ce que le nombre maximal de réparations autorisées soit atteint. Pour ce faire, une clause est choisie aléatoirement parmi l'ensemble des clauses falsifiées par l'interprétation courante (ligne 6). S'il existe une variable appartenant à cette clause permettant de réduire le nombre de clauses falsifiées (descente) cette dernière est

flippée[1] (ligne 8). Sinon, on se trouve dans un minimum local et un critère d'échappement est utilisé (ligne 10). Cette opération est répétée un certain nombre de fois ($MaxFlips$) fixé au départ. Ce processus peut se répéter un nombre maximal de fois ($MaxTries$) fixé au démarrage de la procédure. Un des avantages de cette méthode est la vitesse à laquelle la prochaine variable à flipper est choisie. En effet, le choix de la prochaine variable est limité à un sous-ensemble de variables restreint. De cette manière, WSAT parcourt rapidement de nombreuses interprétations.

3.2 Analyse de conflits et graphe d'implications

Nous introduisons maintenant la notion de graphes d'implications utilisée par les solveurs complets de type CDCL pour analyser les conflits, déduire de nouvelles clauses (ou nogoods) et effectuer un retour arrière non chronologique. Le graphe d'implications est un graphe orienté acyclique (DAG) permettant une représentation de l'affectation des variables et des propagations unitaires issues de ces affectations. Dans un graphe d'implications, chaque sommet représente un littéral et possède un niveau de décision. Dès que l'on affecte une variable, celle-ci est appelée variable de décision et elle donne son rang d'instanciation à toutes les variables qui seront affectées par propagation unitaire résultant de cette affectation. Pour chaque littéral y propagé, on garde en mémoire la clause à l'origine de cette propagation. Cette clause, notée $\overrightarrow{cla}(y)$, est de la forme $(x_1 \vee ... \vee x_n \vee y)$ tel que $\forall x_i, 1 \leq i \leq n, x_i \notin \mathcal{I}$ et $y \in \mathcal{I}$. Lorsqu'un littéral y n'est pas obtenu par propagation unitaire mais qu'il correspond à un point de décision, $\overrightarrow{cla}(y)$ est indéfini et par convention on notera $\overrightarrow{cla}(y) = \bot$.

Lorsque $\overrightarrow{cla}(y) \neq \bot$, on note par $exp(y)$ l'ensemble $\{\overline{x_i} | x_i \in \overrightarrow{cla}(y) \setminus \{y\}\}$, appelé l'ensemble des *explications* de y. Autrement dit, $\overrightarrow{cla}(y) = (x_1 \vee ... \vee x_n \vee y)$ alors les explications sont les littéraux $\overline{x_i}$ qui constituent la condition sous laquelle $\overrightarrow{cla}(y)$ devient une clause unitaire $\{y\}$.

Quand $\overrightarrow{cla}(y)$ est indéfini, $exp(y)$ est égal à l'ensemble vide. Dans un graphe d'implications, chaque noeud admet un ensemble d'explications qui correspond à l'ensemble de ses prédécesseurs dans le graphe. De manière formelle on définit le graphe d'implications de la manière suivante :

Définition 1 (graphe d'implications) *Soit Σ une formule sous forme CNF, $\mathcal{I}_p$ une interprétation partielle. Le graphe d'implications associé à Σ, $\mathcal{I}_p$ et exp est $\mathcal{G}_\Sigma^{\mathcal{I}_p} = (\mathcal{N}, \mathcal{A})$ où :*

1. *$\mathcal{N} = \{x | x \in \mathcal{I}_p\}$ i.e. un noeud pour chaque littéral de $\mathcal{I}_p$;*
2. *$\mathcal{A} = \{(x, y) | x \in \mathcal{I}_p, y \in \mathcal{I}_p, x \in exp(y)\}$.*

[1] Flipper une variable consiste à inverser sa valeur.

Exemple 3.1 *Soient $\Sigma = \{\phi_1, ..., \phi_{11}\}$ une formule CNF tel que :*

$$\begin{array}{ll} \phi_1 : (\overline{x_1} \vee \overline{x_2} \vee \overline{x_3}) & \phi_2 : (x_1 \vee \overline{x_4} \vee \overline{x_5}) \\ \phi_3 : (x_2 \vee \overline{x_1}) & \phi_4 : (x_4 \vee \overline{x_7} \vee \overline{x_6}) \\ \phi_5 : (x_3 \vee \overline{x_5}) & \phi_6 : (x_5 \vee \overline{x_7}) \\ \phi_7 : (x_6 \vee \overline{x_8}) & \phi_8 : (x_7 \vee \overline{x_8}) \\ \phi_9 : (x_8 \vee \overline{x_4}) & \phi_{10} : (x_1 \vee \overline{x_8}) \\ \phi_{11} : (x_7 \vee \overline{x_9}) & \end{array}$$

et $\mathcal{I}_p$ l'interprétation partielle suivante $\mathcal{I}_p = \{\langle (x_2^1) \rangle \langle (x_6^2) \rangle \langle (x_9^3)\ x_7^3\ x_4^3\ x_5^3\ x_3^3\ x_1^3\ \overline{x_1^3} \rangle\}$ où x_i^j indique que le littéral x_i est affecté au niveau j. Les littéraux de décision sont indiqués entre parenthèses et les autres littéraux représentent ceux affectés par propagation unitaire. Le niveau de décision courant est 5 et $\Sigma_{|\mathcal{I}_p} \models_* \bot$. La figure 1 représente le graphe d'implications $\mathcal{G}_\Sigma^{\mathcal{I}_p}$ associée à Σ, $\mathcal{I}_p$ et exp (l'ensemble des explications).

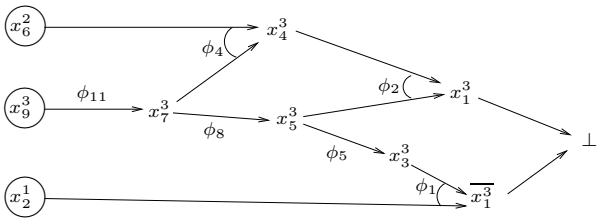


FIG. 1 – Graphe d'implication (exemple 3.1)

Le graphe d'implications est utilisé pour extraire des nogoods. Ceux-ci sont construits par l'application de plusieurs résolutions sur la clause devenue fausse et en remontant ce graphe. Divers types de nogoods peuvent alors être générés. L'un des plus utilisés et des plus efficaces, est le « *first UIP* » (Unique Implication Point). Pour plus de détails, se référer à [18, 1].

4 Analyse de conflits et recherche locale

4.1 Définition des graphes conflits

Nous avons vu dans la section 3.1 qu'un des points importants des méthodes de recherche locales est le critère d'échappement. Nous proposons une nouvelle méthode qui, partant de l'interprétation complète, génère un graphe d'implications. Celui-ci va nous permettre d'ajouter des nogoods

à la formule mais également de choisir les variables à flipper pour sortir de ce minimum local.

Néanmoins, dans le cadre de la recherche locale, le fait de considérer des interprétations complètes rend la définition et la construction de graphes d'implications problèmatique. D'une part, on ne retrouve pas les notions de niveau d'affectation, de littéraux de décision ou propagés. D'autre part, une interprétation complète peut falsifier plusieurs clauses. Dans cette section, nous proposons une nouvelle définition des graphes d'implications adaptée au contexte de la recherche locale stochastique. Nous donnons ci-dessous quelques définitions préalables en considérant une formule CNF Σ et une interprétation complète $\mathcal{I}_c$. On dit qu'un littéral ℓ satisfait (respectivement falsifie) une clause $\beta \in \Sigma$ si $\ell \in \mathcal{I}_c \cap \beta$ (respectivement $\ell \in \mathcal{I}_c \cap \overline{\beta}$). On note l'ensemble des littéraux satisfaisant (resp. falsifiant) une clause β pour une interprétation $\mathcal{I}_c$ par $\mathcal{L}^+_{\mathcal{I}_c}(\beta)$ (respectivement $\mathcal{L}^-_{\mathcal{I}_c}(\beta)$).

Définition 2 (clause unisatisfaite) *Une clause β est dite* unisatisfaite *si $|\mathcal{L}^+_{\mathcal{I}_c}(\beta)| = 1$.*

Définition 3 (clause critique et liée [9]) *Une clause α est dite critique si $|\mathcal{L}^+_{\mathcal{I}_c}(\alpha)| = 0$ et $\forall \ell \in \alpha$, $\exists \alpha' \in \Sigma$ avec $\sim \ell \in \alpha'$ et $|\mathcal{L}^+_{\mathcal{I}_c}(\alpha')| = 1$ (α' est satisfaite en $\sim \ell$). Les clauses α' sont dites* liées *à α pour l'interprétation $\mathcal{I}_c$.*

Exemple 4.1 *Soient $\Sigma = (\overline{a} \vee \overline{b} \vee \overline{c}) \wedge (a \vee \overline{b}) \wedge (b \vee \overline{c}) \wedge (c \vee \overline{a})$ et $\mathcal{I}_c = \{a, b, c\}$. La clause $\alpha_1 = (\overline{a} \vee \overline{b} \vee \overline{c})$ est critique. Les trois autres clauses sont liées à α_1 pour $\mathcal{I}_c$.*

Il est important de noter que dans un minimum local toutes les clauses falsifiées sont critiques [9]. On peut maintenant définir la notion de graphe conflit pour les interprétations complètes.

Définition 4 (graphe conflit sur z) *Soient Σ une formule sous forme CNF, $\mathcal{I}$ une interprétation complète conflictuelle de Σ. Étant données deux clauses de Σ, $\beta = \{\beta_1, \ldots, \beta_k, z\}$ falsifiée par $\mathcal{I}$ et $\gamma = \{\gamma_1, \ldots, \gamma_l, \overline{z}\}$ unisatisfaite en z pour $\mathcal{I}$, on construit le graphe conflits $\mathcal{G}^z_{\Sigma,\mathcal{I}} = (\mathcal{N}, \mathcal{A})$ de la manière suivante :*

1. – $\{z, \overline{z}, \bot\} \subseteq \mathcal{N}$;
   – $\{\overline{\gamma_1}, \ldots, \overline{\gamma_l}\} \subseteq \mathcal{N}$;
   – $\{\overline{\beta_1}, \ldots, \overline{\beta_k}\} \subseteq \mathcal{N}$;
2. – $\{(\underline{z}, \bot), (\overline{z}, \bot)\} \subseteq \mathcal{A}$;
   – $\{(\overline{\beta_1}, z), \ldots, (\overline{\beta_k}, z)\} \subseteq \mathcal{A}$;
   – $\{(\overline{\gamma_1}, \overline{z}), \ldots, (\overline{\gamma_k}, \overline{z})\} \subseteq \mathcal{A}$;
3. *$\forall x \in \mathcal{N}$, si $x \neq z$ et $\alpha = \bigwedge\{y \in \mathcal{N} \mid (y, x) \in \mathcal{A}\} \not\models \bot$ alors $\overline{\alpha} \vee x \in \Sigma$ et est unisatisfaite en x.*

Il est clair que pour un littéral z donné, les clauses β et γ ne sont pas uniques. De plus, il est possible que plusieurs clauses unisatisfaites expliquent un même littéral. On peut donc remarquer qu'il n'y a pas unicité du graphe conflit.

Exemple 4.2 *Considérons de nouveau l'exemple 3.1. Soit l'interprétation complète $\mathcal{I}_c \subseteq \mathcal{V}_\Sigma$ tel que $\mathcal{I}_c = \{x_1, \ldots, x_9\}$. La figure 2 schématise un graphe conflit $\mathcal{G}^{x_1}_{\Sigma,\mathcal{I}}$ construit sur la variable x_1.*

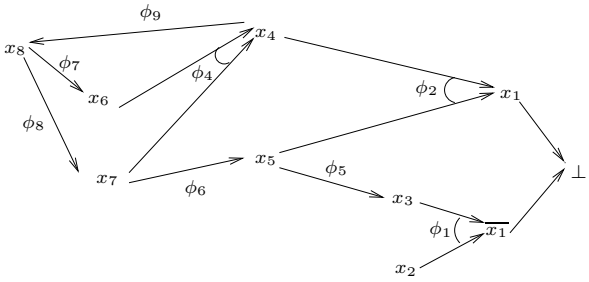


FIG. 2 – Graphe conflit défini sur la variable x_1

Il est important de noter que l'existence d'un tel graphe n'est pas toujours assurée, et la proposition suivante exprime les conditions sous lesquelles la construction de ce graphe conflit est possible.

Proposition 1 *Soient Σ une formule sous forme CNF, $\mathcal{I}$ une interprétation complète construite sur les variables de Σ. Le graphe conflit $\mathcal{G}^x_{\Sigma,\mathcal{I}}$ est constructible si et seulement si x et $\overline{x}$ apparaissent respectivement dans une clause fausse et une clause unisatisfaite.*

Preuve 1 *Cette proposition découle directement de la définition d'un graphe conflit.*

Corrolaire 2 *Soit $\alpha \in \Sigma$ une clause critique. $\forall x \in \alpha$ il est possible de construire un graphe conflit défini sur x.*

Preuve 2 *Par définition d'une clause critique $\alpha \in \Sigma$ on a $\forall x \in \alpha$, $\exists \beta \in \Sigma$ unisatisfaite en x. D'après la proposition 1, il est facile de voir que $\forall x \in \alpha$ il existe un graphe conflit en x.*

Lorsque l'on se trouve dans un minimum local, toutes les clauses non satisfaites sont critiques. Le corrolaire 2 nous assure donc que, dans ce cas, il est toujours possible de créer un graphe conflit. Ceci est très important, puisque cela va nous permettre de construire un graphe afin de générer un nogood qui va lui même permettre de s'extraire du minimum local. Néanmoins, générer de tels nogoods en

partant d'un graphe conflit est un problème difficile. En effet, la notion de niveau étant absente, l'analyse de conflit classique basée sur la notion d'UIPs ne peut être étendue. De plus, les graphes conflits, tel que définis précédemment, peuvent contenir des cycles. La présence de cycles au sein d'un graphe conflit peut conduire à produire par résolution des clauses tautologiques. Ces clauses sont par nature inutiles. Pour palier à ces problèmes, nous proposons dans la section suivante, une méthode, qui, à partir du graphe conflit d'une interprétation complète, extrait le graphe d'implications associé. De cette manière, on retrouve le cas classique des graphes d'implications utilisés dans les solveurs SAT modernes qui permettent d'extraire facilement et rapidement des nogoods importants pour simplifier la recherche.

4.2 Construction des graphes conflits

Afin de palier aux problèmes énoncés dans la section précédente, nous proposons une première méthode basée sur la propagation unitaire. Partant de l'interprétation complète conflictuelle $\mathcal{I}$, nous allons construire une interprétation partielle par propagation unitaire dont les points de choix ont les mêmes valeurs dans les deux interprétations. Cette interprétation, que nous définissons ci-dessous, permettra ensuite de définir un graphe conflit acyclique.

Définition 5 (interprétation partielle dérivée) *Soient Σ une formule sous forme CNF, $\mathcal{I}$ une interprétation complète conflictuelle construite sur $\mathcal{V}_\Sigma$. L'interprétation partielle dérivée de $\mathcal{I}$, noté $\mathcal{I}'$, est construite de manière incrémentale de la façon suivante :*

- $\mathcal{I}'_0 = \emptyset$;
- $\mathcal{I}'_{i+1} = \mathcal{I}'_i \cup \{(x_{i+1}), x^1_{i+1}, ..., x^k_{i+1}\}$ *tel que* $x_{i+1} \in \mathcal{V}_\Sigma \setminus \mathcal{V}_{\mathcal{I}'_i}$ *et* $\forall j, 1 \leqslant j \leqslant k$ *on a* $\Sigma_{|_{\mathcal{I}'_i \cup \{x_{i+1}\}}} \models_* x^j_{i+1}$ *avec* $x^j_{i+1} \in \mathcal{I}$;
- $\mathcal{I}' = \mathcal{I}'_i \cup \{(x_{i+1}), x^1_{i+1}, ..., x^l_{i+1}\}$ *tel que* $x_{i+1} \in \mathcal{V}_\Sigma \setminus \mathcal{V}_{\mathcal{I}'_i}$ *et* $\forall j, 1 \leqslant j \leqslant l$ *on a* $\Sigma_{|_{\mathcal{I}'_i \cup \{x_{i+1}\}}} \models_* x^j_{i+1}$ *avec* $x^j_{i+1} \in \mathcal{I}$ *pour* $j \neq l$ *et* $x^l_{i+1} \notin \mathcal{I}$.

L'ensemble des points de choix sera nommé ensemble conflit et on appellera littéral conflit le littéral $x \in \mathcal{I}' \setminus \mathcal{I}$.

Le choix des variables apparaissant dans l'ensemble conflit est un choix heuristique et conduit à différentes interprétations partielles dérivées.

Exemple 4.3 *Reprenons la formule Σ de l'exemple 3.1 et l'interprétation complète et conflictuelle $\mathcal{I}_c = \{x_1, \dots x_9\}$.*

Dans un premier temps, considérons l'ordre lexicographique pour générer l'interprétation partielle dérivée. On obtient :

- $\mathcal{I}'_0 = \emptyset$
- $\mathcal{I}'_1 = \{(x_1), x^1_2, \overline{x^1_3}\}$

La variable conflit est alors x_3 et l'ensemble conflit est limité $\{x_1\}$. Si nous considérons maintenant l'ordre lexicographique inverse, alors l'interprétation partielle dérivée est :

- $\mathcal{I}'_0 = \emptyset$
- $\mathcal{I}'_1 = \{(x^1_9), x^1_7, x^1_5, x^3_1\}$
- $\mathcal{I}'_2 = \{(x^1_9), x^1_7, x^1_5, x^3_1, (x^2_8), x^2_6, x^2_4, x^2_1, x^2_2, \overline{x^2_2}\}$

Le littéral conflit est x_2 et on a comme ensemble conflit $\{x_9, x_8\}$.

L'interprétation complète étant conflictuelle, son interprétation partielle dérivée va différer sur au moins un littéral (le littéral conflit). La proposition suivante exprime le fait qu'il existe au moins une clause falsifiée contenant ce littéral. C'est à partir de celui-ci que l'on construira notre graphe conflit.

Proposition 3 *Soient Σ une formule CNF, $\mathcal{I}$ une interprétation complète conflictuelle de Σ et $\mathcal{I}'$ une interprétation partielle dérivée de $\mathcal{I}$. Soit x le littéral conflit, alors $exp(x) \subseteq \mathcal{I}$ et la clause $\overrightarrow{cla}(x)$ est falsifiée par $\mathcal{I}$.*

Preuve 3 *Tout d'abord il est facile de voir que par construction de $\mathcal{I}'$ on a $exp(x) \subseteq \mathcal{I}$. En effet, supposons que $exp(x) \not\subseteq \mathcal{I}$ alors $\exists y \in exp(x)$ tel que $y \notin \mathcal{I}$. Par définition $exp(x) \subseteq \mathcal{I}'$, donc $y \in \mathcal{I}'$ et par conséquent le littéral y est aussi un littéral conflit. Par construction de $\mathcal{I}'$ il ne peut y avoir qu'un seul littéral conflit, il en résulte alors $y = x$. Cette conclusion est absurde car une variable propagée ne peut pas appartenir à son explication.*

Montrons à présent que $\mathcal{I} \not\models \overrightarrow{cla}(x)$. Pour cela, raisonnons également par l'absurde et supposons $\overrightarrow{cla}(x)$ soit satisfait par $\mathcal{I}$. En premier lieu, on peut noter que x est une variable obtenue par propagation unitaire. Il est donc possible de trouver une explication $exp(x)$ et une clause $\overrightarrow{cla}(x) \in \Sigma$ tel que $\overrightarrow{cla}(x) = \overline{exp(x)} \vee x$. On sait par ailleurs que $exp(x) \subseteq \mathcal{I}$, donc $\overline{exp(x)} \not\subseteq \mathcal{I}$. Puisque $\overrightarrow{cla}(x)$ est satisfaite sous l'interprétation $\mathcal{I}$, elle ne peut l'être qu'en x. Ce qui est absurde puisque $x \notin \mathcal{I}$.

La proposition suivante exprime le fait que toutes les clauses (hormis la clause falsifiée) ayant servi à la construction du graphe sont unisatisfaites.

Proposition 4 *Soient Σ une formule CNF, $\mathcal{I}$ une interprétation complète conflictuelle de Σ, $\mathcal{I}'$ une*

interprétation partielle dérivée de $\mathcal{I}$ en fonction de Σ et x le littéral conflit associé à $\mathcal{I}'$. Considérons le graphe d'implications $\mathcal{G}_\Sigma^{\mathcal{I}'} = (\mathcal{N}, \mathcal{A})$, alors $\forall y \in \mathcal{N} \setminus \{x\}$ on a $\overrightarrow{cla}(y) = \bot$ ou $\overrightarrow{cla}(y)$ est unisatisfaite par $\mathcal{I}'$ en y.

Preuve 4 *Deux cas sont à considérer :*

1. *y est un point de choix et alors $\overrightarrow{cla}(y) = \bot$;*
2. *y n'est pas un littéral propagé. Il existe $\overrightarrow{cla}(y) \in \Sigma$ tel que $\overrightarrow{cla}(y) = \overline{exp(y)} \vee y$. Par construction de $\mathcal{I}'$, on a $x \notin exp(y)$ et $\mathcal{I}' \setminus \{x\} \subset \mathcal{I}$. Puisque $y \neq x$ on a $\{exp(y), y\} \subseteq \mathcal{I}' \setminus \{x\}$. Par transitivité on a $\{exp(y), y\} \subseteq \mathcal{I} \setminus \{x\}$. Donc la clause $\overrightarrow{cla}(y)$ est unisatisfaite par $\mathcal{I}$ en y.*

Nous prouvons dans la proposition 5 que le graphe d'implications obtenu à partir de l'interprétation partielle dérivée peut être étendu en un graphe conflit sur le littéral conflit. Ayant maintenant un graphe d'implications comme ceux utilisés dans les solveurs CDCL classiques [5], il est possible d'analyser celui-ci pour générer un nogood qui sera ensuite ajouté à la base de clauses.

Proposition 5 *Soient Σ une formule CNF, $\mathcal{I}$ une interprétation complète de Σ, $\mathcal{I}'$ une interprétation partielle dérivée et x le littéral conflit associé à $\mathcal{I}'$. Si $\exists \alpha \in \Sigma$ unisatisfaite par $\mathcal{I}$ en x, alors il est possible d'étendre le graphe d'implication $\mathcal{G}_\Sigma^{\mathcal{I}'} = (\mathcal{N}, \mathcal{A})$ associé à $\mathcal{I}'$ en un graphe conflit $\mathcal{G}^x_{\mathcal{I},\Sigma} = (\mathcal{N}', \mathcal{A}')$ de la manière suivante :*

- *$\mathcal{N}' = \mathcal{N} \cup \{y \in \overline{\alpha} \setminus x\} \cup \{\overline{x}, \bot\}$;*
- *$\mathcal{A}' = \mathcal{A} \cup \{(y, \overline{x}) | y \in \overline{\alpha} \setminus x\} \cup \{(x, \bot), (\overline{x}, \bot)\}$.*

Preuve 5 *Avant de vérifier que $\mathcal{G}^x_{\mathcal{I},\Sigma}$ est bien un graphe conflit valide, il faut identifier les clauses $\beta = \{\beta_1, \ldots, \beta_k, z\} \in \Sigma$ falsifiée par $\mathcal{I}$ et $\gamma = \{\gamma_1, \ldots, \gamma_l, \overline{z}\} \in \Sigma$ unisatisfaite en z pour $\mathcal{I}$. Par hypothèse, la clause α est unisatisfaite par $\mathcal{I}$ en x. On peut donc poser $\gamma = \alpha$. D'après la proposition 3, nous pouvons prendre pour β la clause $\overrightarrow{cla}(x)$. En effet, $x \in \overrightarrow{cla}(x)$ et la clause $\overrightarrow{cla}(x)$ est bien falsifiée par $\mathcal{I}$. Les deux clauses servant à la construction du graphe conflit étant identifiées, pour valider que $\mathcal{G}^x_{\mathcal{I},\Sigma} = (\mathcal{N}', \mathcal{A}')$ est un graphe conflit, il suffit, d'après la définition 4, de vérifier les propriétés suivantes :*

1. *– $\{x, \overline{x}, \bot\} \subseteq \mathcal{N}'$. Par hypothèse, on a $\{\overline{x}, \bot\} \subseteq \mathcal{N}'$, reste à monter que $x \in \mathcal{N}'$. On sait que $x \in \mathcal{I}'$, par construction du graphe $\mathcal{G}_\Sigma^{\mathcal{I}'}$, on a $x \subseteq \mathcal{N}$. Donc, puisque $\mathcal{N} \subseteq \mathcal{N}'$, $x \in \mathcal{N}'$;*
   – $\{\overline{\gamma_1}, \ldots, \overline{\gamma_l}\} \subseteq \mathcal{N}'$. On a $\{y \in \overline{\alpha} \setminus x\} \subseteq \mathcal{N}'$ et $\gamma = \alpha$. Donc $\{y \in \overline{\gamma} \setminus x\} \subseteq \mathcal{N}'$;
   – $\{\overline{\beta_1}, \ldots, \overline{\beta_k}\} \subseteq \mathcal{N}'$. Par hypothèse, $\beta = \overrightarrow{cla}(x) = \beta_1 \vee \cdots \vee \beta_k \vee x = \overline{exp}(x) \vee x$. D'où $exp(x) = \{\overline{\beta_1}, \ldots, \overline{\beta_k}\}$. D'après la proposition 3, on a $exp(x) \subseteq \mathcal{I}'$, d'où $exp(x) \subseteq \mathcal{N}$ (voir définition 1). Puisque $\mathcal{N} \subseteq \mathcal{N}'$, par transitivité on a $exp(x) \subseteq \mathcal{N}'$;
2. *– $\{(x, \bot), (\overline{x}, \bot)\} \subseteq \mathcal{A}'$. Par construction ;*
   – $\{(\overline{\gamma_1}, \overline{x}), \ldots, (\overline{\gamma_k}, \overline{x})\} \subseteq \mathcal{A}'$. Idem ;
   – $\{(\overline{\beta_1}, x), \ldots, (\overline{\beta_k}, x)\} \subseteq \mathcal{A}'$. On sait que $exp(x) = \{\overline{\beta_1}, \ldots, \overline{\beta_k}\}$ et que $\{exp(x), x\} \subseteq \mathcal{I}'$. Par construction de $\mathcal{A}'$ et d'après la définition 1, on a $\{(\overline{\beta_1}, x), \ldots, (\overline{\beta_k}, x)\} \subseteq \mathcal{A} \subseteq \mathcal{A}'$;
3. *$\forall x \in \mathcal{N}$, si $x \neq z$ et $\alpha = \bigwedge\{y \in \mathcal{N} \mid (y, x) \in \mathcal{A}\} \not\models \bot$ alors $\overline{\alpha} \vee x \in \Sigma$ et est unisatisfaite en x. D'après la proposition 4, $\forall y \in \mathcal{N}$ tel que $y \neq x$ et $\overrightarrow{cla}(y) \neq \bot$ on a $\overrightarrow{cla}(y)$ unisatisfaite par $\mathcal{I}$ en y. Par construction de $\mathcal{G}_\Sigma^{\mathcal{I}}$ et $\mathcal{G}^x_{\mathcal{I},\Sigma}$ la propriété précédente est vérifiée.*

Exemple 4.4 *Reprenons l'exemple 3.1 et les interprétations partielles dérivées obtenues dans l'exemple 4.3. Nous pouvons alors étendre les graphes d'implications associés à ces deux interprétations en deux graphes conflits schématisés dans les figures 3 (ordre lexicographique) et 4 (ordre lexicographique inverse).*

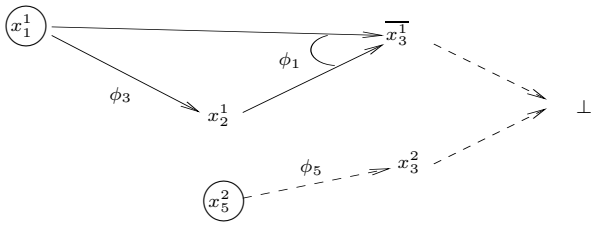


FIG. 3 – Graphe conflit construit à l'aide de la propagation unitaire (ordre lexicographique)

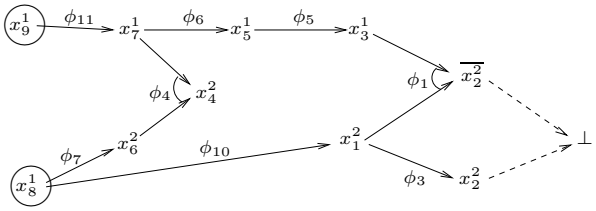


FIG. 4 – Graphe conflit construit à l'aide de la propagation unitaire (ordre lexicographique inverse)

4.3 Implantation

Nous proposons d'intégrer le graphe conflit défini dans la section précédente au sein d'un solveur de type WSAT, le graphe conflit sera construit

Algorithme 2 : CDLS

Input : Σ une formule CNF
Output : SAT ou $UNSAT$ si la satisfaisabilité de Σ a pu être décidée, $UNKNOWN$ sinon

1 **for** $i \leftarrow 1$ *to* $MaxTries$ **do**
2 $\mathcal{I}_c \leftarrow$ interpretationCompletePU(Σ);
3 **for** $j \leftarrow 1$ *to* $MaxFlips$ **do**
4 **if** $\mathcal{I}_c \models \Sigma$ **then**
5 **return** SAT;
6 $\Gamma = \{\alpha \in \Sigma | \ \mathcal{I}_c \not\models \alpha\}$;
7 **if** $\exists x \in \mathcal{V}_\Gamma$ *permettant une descente* **then**
8 $flipper(x)$;
9 **else**
 `/* minimum local */`
10 $\alpha \in \Gamma$;
11 $\beta \leftarrow analyseConflitRL(\Sigma, \mathcal{I}_c, \alpha)$;
12 **if** $\beta = \bot$ **then**
13 **return** UNSAT;
14 $\Sigma \leftarrow \Sigma \cup \{\beta\}$;
15 **return** $UNKNOWN$;

Fonction `analyseConflitRL`

Input : - Σ une formule CNF ;
- $\mathcal{I}_c$ une interprétation complète ;
- $\alpha \in \Sigma$ une clause critique pour$\mathcal{I}_c$.

Output : β une clause construite sur $\mathcal{V}_\Sigma$

1 $\mathcal{E} \leftarrow ensembleConflit(\Sigma, \mathcal{I}_c, \alpha)$;
2 $\gamma \leftarrow \emptyset$;
3 $\mathcal{I}_p \leftarrow \emptyset$;
4 **while** $(\gamma = \emptyset)$ *and* $(\mathcal{I}_p \subset \mathcal{I}_c)$ **do**
5 $\mathcal{E} \leftarrow \mathcal{E} \setminus \mathcal{I}_p$;
6 $\mathcal{I}_p \leftarrow \mathcal{I}_p \cup \{x\}$ tel que $x \in \mathcal{E}$;
7 $\gamma \leftarrow BCP()$;
8 **if** $\gamma \neq \emptyset$ **then** `/* CAS 1 */`
9 $\beta \leftarrow analyseConfitClassique(\mathcal{G}_\Sigma^{\mathcal{I}_p})$;
10 $flipper(x)$ avec x littéral assertif;
11 **else** `/* CAS 2 */`
12 $\mathcal{I}' \leftarrow$ interprétation dérivé associée à $\mathcal{I}_p$;
13 $y \leftarrow$ littéral conflit de $\mathcal{I}'$;
14 $\mathcal{G}^y_{(\Sigma,\mathcal{I}')} \leftarrow$ graphe conflit étendu de $\mathcal{G}^{\mathcal{I}'}_\Sigma$;
15 $\beta \leftarrow analyseConfitClassique(\mathcal{G}^y_{(\Sigma,\mathcal{I}')})$;
16 **forall** $x \in \mathcal{I}_p \setminus \mathcal{I}_c$ **do**
17 $flipper(x)$;
18 **return** β;

lorsque WSAT aura atteint un minimum local afin de s'échapper de celui-ci. Cette méthode est nommée CDLS (*Conflict Driven for Local Search*).

Lors de la construction de l'interprétation partielle dérivée, deux cas peuvent se présenter. Soit un conflit est atteint lors de la propagation unitaire (voir figure 4) et dans ce cas, le graphe d'implications résultant est utilisé pour analyser le conflit comme le font les solveurs SAT modernes et extraire une clause assertive associée au premier UIP. Sinon (voir figure 3), nous étendons le graphe d'implications de l'interprétation partielle dérivée en un graphe conflit sur le littéral conflit, et de la même manière, on extrait une clause assertive. Dans les deux cas, une clause assertive est ajoutée à la base de clauses de la formule. Lorsque celle-ci est égale à la clause vide l'insatisfaisabilité de la formule est ainsi démontrée. Contrairement à WSAT classique qui n'autorise qu'un seul flip à chaque étape, CDLS va pouvoir flipper un ensemble de variables afin de s'échapper d'un minimum local. Ces variables sont celles qui diffèrent entre l'interprétation complète et l'interprétation partielle dérivée.

Il est important de noter que l'algorithme CDLS n'est pas un algorithme hybride comme [6, 7]. C'est une méthode de recherche locale stochastique. La propagation unitaire est uniquement utilisée pour construire et analyser le graphe conflit et ainsi générer des nogoods.

L'algorithme CDLS (algorithme 2) prend en paramètre une CNF Σ et peut retourner trois valeurs, SAT, $UNSAT$ ou $UNKNOWN$. L'algorithme reprend le schéma de base de WSAT (voir algorithme 1). Notons que les interprétations initiales sont construites en utilisant la propagation unitaire (ligne 2 de l'algorithme CDLS et fonction `interpretationCompletePU`). En effet, l'utilisation de la propagation unitaire pour initialiser l'interpération courante de WSAT permet de s'approcher plus rapidement d'un minimum local et de prendre en compte certaines dépendances fonctionnelles entre les variables. Tant qu'une descente est possible on flippe une variable permettant de diminuer le nombre de clauses fausses. Une fois, un minimum local atteint, on analyse le conflit comme expliqué précédemment afin de générer un nogood β qui est ajouté à la base de clauses. La fonction `analyseConflitRL` est au coeur de notre contribution. Cette fonction commence par sélectionner un ensemble des variables conflits qui vont nous permettre de construire une interprétation partielle dérivée conduisant à un conflit. C'est la fonction `ensembleConflit` qui s'en charge. Elle choisit les variables dans les clauses liées à la clause falsifiée α, afin de forcer l'interprétation partielle qui va en être déduite, à rester dans le voisinage de α. La fonction `BCP` effectue la propagation unitaire. Elle renvoie une clause falsifiée si la propagation uni-

Fonction `interpretationCompletePU`

Input : Σ une formule CNF
Output : $\mathcal{I}_c$ une interprétation complète de Σ

1 $\Sigma' \leftarrow \Sigma$;
2 $\mathcal{I}_c \leftarrow \emptyset$;
3 **while** $|\mathcal{I}_c| \neq |\mathcal{V}_\Sigma|$ **do**
4 $\mathcal{I}_c \leftarrow \mathcal{I}_c \cup \{x | x \in \mathcal{V}_\Sigma \setminus \mathcal{I}_c\}$;
5 $\mathcal{P} \leftarrow \{x\} \cup \{y |\ \Sigma'_{|_x} \models_* y\}$;
6 **foreach** $y \in \mathcal{P}$ **do**
7 **if** $\overline{y} \in \mathcal{I}_c$ **then**
8 $\mathcal{P} \leftarrow \mathcal{P} \setminus \{y\}$;
9 $\mathcal{I}_c \leftarrow \mathcal{I}_c \cup \mathcal{P}$;
10 $\Sigma' \leftarrow \Sigma'_{|\mathcal{P}}$;
11 **return** $\mathcal{I}_c$;

Fonction `ensembleConflit`

Input : - Σ une formule CNF ;
- $\mathcal{I}_c$ une interprétation complète ;
- $\alpha \in \Sigma$ une clause critique pour $\mathcal{I}_c$.
Output : $\mathcal{C}$ un ensemble de littéraux conflit

1 $\mathcal{C} \leftarrow \emptyset$;
2 **forall** $x \in \overline{\alpha}$ **do**
3 $\beta \in \Sigma$ unisatisfaite en x;
4 $\mathcal{C} \leftarrow \mathcal{C} \cup \{\beta \setminus \{x\}\}$;
5 **return** $\mathcal{C}$;

taire conduit à un conflit et la clause vide sinon. C'est cette fonction qui permet de construire ensuite l'interprétation partielle $\mathcal{I}_p$ (ligne 4-7). L'interprétation partielle conduit donc, soit à un conflit par propagation unitaire, soit à une discordance entre l'interprétation partielle et l'interprétation complète. Dans le premier cas, une analyse de conflit classique est effectuée (ligne 9) et le littéral assertif est flippé. Dans le second, on extrait l'interprétation partielle dérivée de $\mathcal{I}_p$ et on génère le graphe conflit associé (ligne 12-14 et proposition 5). À partir de là, il est possible d'analyser le conflit. Toutes les variables qui diffèrent de valeur entre l'interprétation partielle et l'interprétation complète sont alors flippées.

5 Expérimentations

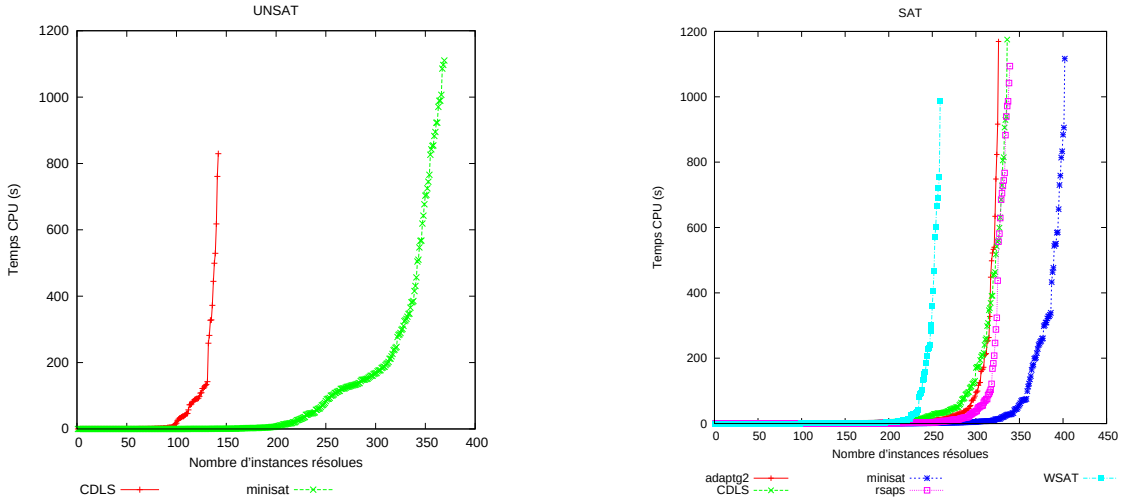
Les résultats expérimentaux reportés dans cette section ont été obtenus sur un Xeon 3.2 GHz (2 Go RAM). Le temps CPU est limité à 1200 secondes et les résultats sont reportés en secondes. Nous comparons CDLS à 3 méthodes incomplètes : WSAT [16], rsaps [11] et adaptg2 [12], et à minisat [5] un des solveurs complets CDCL les plus efficaces. Les instances utilisées sont issues des dernières compétitions SAT et sont divisées en trois catégories : *crafted* (1439 instances), *industrial* (1305) et *random* (2172). Toutes les instances sont prétraitées à l'aide de SATElite [4].

La figure 5 montre, pour chaque catégorie d'instances et en fonction de la satisfaisabilité de celles-ci, le nombre d'instances résolues en fonction du temps. Rappelons, tout d'abord que adaptg2, rsaps et WSAT sont des méthodes qui ne peuvent répondre qu'à la satisfaisabilité d'une instance. En ce qui concerne la catégorie *crafted*, le solveur minisat mis à part, notre approche est compétitive et résoud sensiblement le même nombre d'instances que les approches de recherche locale récentes adaptg2 et rsaps. Deux autres points sont à noter. D'une part, CDLS résoud beaucoup plus d'instances SAT que WSAT sur lequel il est basé. De l'autre, CDLS est capable de résoudre un grand nombre d'instances UNSAT.

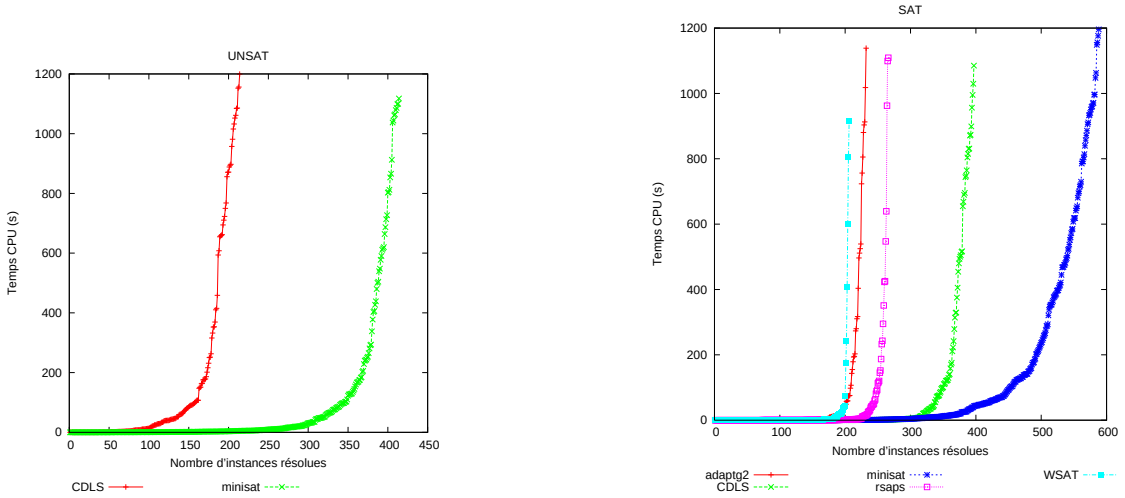
Pour les instances de la catégorie *industrial*, CDLS résoud deux fois plus d'instances SAT que les solveurs stochastiques classiques et est capable de résoudre près de 160 instances UNSAT. L'analyse des conflits dans le cadre de la recherche locale permet donc de résoudre efficacement des instances structurées, qu'elles soient satisfaisables ou pas.

Enfin, concernant la catégorie *random*, notons, tout d'abord, que notre méthode n'arrive pas à prouver l'insatisfaisabilité pour les instances de cette catégorie. Nous pensons que cela provient du choix heuristique de l'ensemble conflit (voir fonction `ensembleConflit`). Il semble que CDLS ne se focalise pas assez sur le même sous-ensemble inconsistant. Au niveau des instances random satisafaisables, CDLS est le moins efficace des solveurs stochastiques. Ceci peut s'expliquer par le fait que la construction du graphe conflit est coûteuse en temps et qu'elle ralentit donc notre méthode. De ce fait, CDLS parcourt moins d'interprétations que les autres méthodes de recherche locale et a donc moins de chance de trouver un modèle. Notons également les mauvaises performances de minisat sur ce type d'instances.

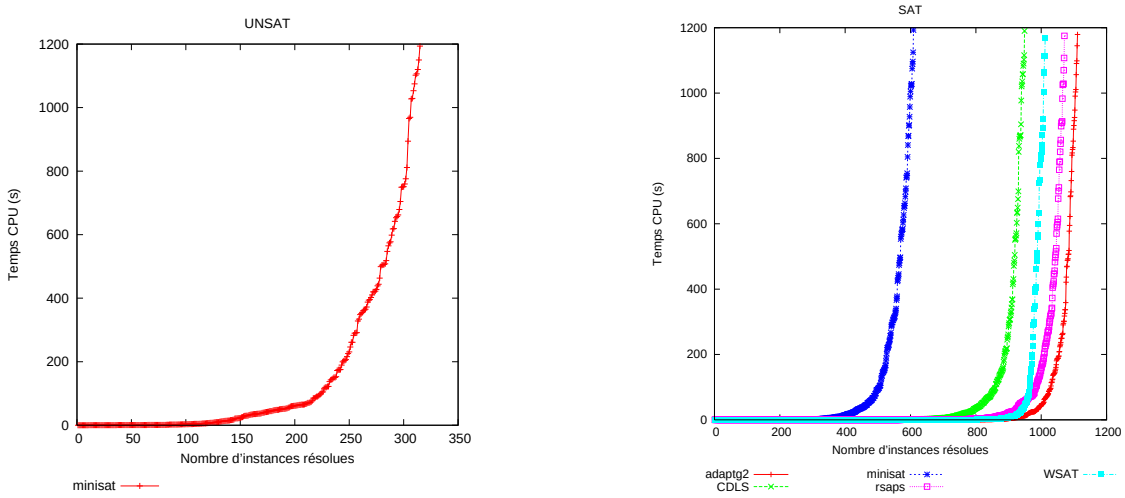
Globalement, CDLS est plus efficace que les autres solveurs basés sur la recherche locale. Il améliore grandement les performances de WSAT sur lequel il est basé. Il serait donc intéressant de greffer notre analyse de conflits à d'autres types d'algorithme de recherche locale, comme rsaps, afin d'améliorer leur performances. Il reste que CDLS ne rivalise pas encore avec les approches CDCL comme minisat dans les catégories crafted et industriels. Néanmoins, les premiers résultats obtenus sont extrêmement encourageants.



catégorie CRAFTED



catégorie INDUSTRIAL



catégorie RANDOM

FIG. 5 – Nombre d'instances résolues en fonction du temps

6 Conclusion

Dans cet article nous proposons une nouvelle approche pour sortir des minimums locaux dans le cadre des méthodes de recherche locale pour SAT. Notre approche est basée sur une extension de l'analyse des conflits utilisé par les solveurs SAT modernes complets. L'objectif est double : améliorer les méthodes de recherche locale sur les problèmes structurés et répondre à un des challenges les plus importants de la communauté SAT, à savoir, proposer une méthode incomplète efficace répondant à l'insatisfaisabilité d'une formule.

Les premiers résultats obtenus sur un large panel d'instances sont très encourageants. Notre solveur CDLS résout beaucoup d'instances insatisfaisables, de plus, il est beaucoup plus efficace que les autres méthodes de recherche pour résoudre des problèmes industriels. Dans les travaux futurs, nous comptons améliorer l'heuristique de choix des variables appartenant à l'ensemble conflit. Nous souhaitons également analyser et extraire les nogoods sans avoir recours à la propagation unitaire. Enfin, l'utilisation du premier UIP n'est pas primordial ici, et nous allons étudier diverses pistes pour choisir le meilleur nogood à extraire.

Références

[1] Gilles Audemard, Lucas Bordeaux, Youssef Hamadi, Saïd Jabbour, and Lakhdar Saïs. Un cadre général pour l'analyse de conflits. Dans *Journées Francophones de Programmation par Contraintes*, pages 267–276, 2008.

[2] Byungki Cha and Kazuo Iwama. Adding new clauses for faster local search. Dans *AAAI/IAAI, Vol. 1*, pages 332–337, 1996.

[3] Martin Davis, George Logemann, and Donald W. Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7) :394–397, 1962.

[4] N. Eén and A. Biere. Effective preprocessing in SAT through variable and clause elimination. Dans *SAT*, pages 61–75, 2005.

[5] Niklas Eén and Niklas Sörensson. An extensible sat-solver. Dans *SAT*, pages 502–518, 2003.

[6] Lei Fang and Michael S. Hsiao. A new hybrid solution to boost sat solver performance. Dans *2007 Design, Automation and Test in Europe Conference and Exposition (DATE 2007)*, pages 1307–1313, 2007.

[7] Eugene Goldberg. A decision-making procedure for resolution-based sat-solvers. Dans *SAT*, pages 119–132, 2008.

[8] Eugene Goldberg and Yakov Novikov. Berkmin : A fast and robust sat-solver. *Discrete Applied Mathematics*, 155(12) :1549–1561, 2007.

[9] Eric Gregoire, Bertrand Mazure, and Cedric Piette. Extracting muses. Dans *European Conference on Artificial Intelligence (ECAI'06)*, pages 387–391, Trento (Italy), August 2006.

[10] Edward A. Hirsch and Arist Kojevnikov. Unitwalk : A new sat solver that uses local search guided by unit clause elimination. *Ann. Math. Artif. Intell.*, 43(1) :91–111, 2005.

[11] Frank Hutter, Dave A. D. Tompkins, and Holger H. Hoos. Scaling and probabilistic smoothing : Efficient dynamic local search for sat. Dans *CP*, pages 233–248, 2002.

[12] Chu Min Li, Wanxia Wei, and Harry Zhang. Combining adaptive noise and look-ahead in local search for sat. Dans *SAT*, pages 121–133, 2007.

[13] Bertrand Mazure, Lakhdar Saïs, and Eric Grégoire. Tabu search for sat. Dans *AAAI*, pages 281–285, juillet 1997.

[14] Lakhdar Sais, editeur. *Problème SAT : Progrès et Défis*, chapitre 6. Hermes, 2008.

[15] Bart Selman and Henry A. Kautz. An empirical study of greedy local search for satisfiability testing. Dans *AAAI*, pages 46–51, 1993.

[16] Bart Selman, Henry A. Kautz, and Bram Cohen. Noise strategies for improving local search. Dans *AAAI*, pages 337–343, 1994.

[17] Bart Selman, Henry A. Kautz, and David A. McAllester. Ten challenges in propositional reasoning and search. Dans *IJCAI (1)*, pages 50–54, 1997.

[18] Lintao Zhang, Conor F. Madigan, Matthew W. Moskewicz, and Sharad Malik. Efficient conflict driven learning in boolean satisfiability solver. Dans *ICCAD*, pages 279–285, 2001.

RB-SAT: Un nouveau modèle SAT basé sur les codages SAT du modèle RB

Haibo Huang **Chu Min Li** **Nouredine Ould Mohamedou*** **Ke Xu**

MIS, Universit de Picardie Jules Verne
5 Rue du Moulin Neuf 80000 Amiens, France.
National Lab of Software Development Environment,
Beihang University, Beijing 100083, China

chu-min.li@u-picardie.fr,wavebywind@163.com, kexu@nlsde.buaa.edu.cn

Résumé

La génération d'instances SAT difficiles est de grande importance à la fois théorique et pratique. Le Modèle RB est un modèle CSP dont les points de seuil peuvent être précisément localisés et l'on peut garantir que les instances générées à ce seuil seront difficiles. Dans ce papier, nous présentons trois différentes méthodes pour coder des instances CSP, à partir d'un Modèle RB, en instances SAT. Ensuite, nous effectuons systématiquement des comparaisons et analyses détaillées. Grâce aux résultats expérimentaux, nous avons constaté que, même en utilisant des façons différentes de coder ces instances, celles-ci demeurent assez difficiles et le pic du coût de la recherche de solutions pour ces instances correspond exactement à celui des instances CSP, ce qui signifie que les instances SAT générées de cette façon ont hérité des bonnes caractéristiques du Modèle RB. Nous avons finalement choisi la méthode la plus naturelle et « directe » de codage pour définir un modèle SAT simple appelé RB-SAT. Avec son aisance de compréhension et d'utilisation, RB-SAT est ainsi un autre modèle SAT en plus de *k*-SAT aléatoire pour étudier la difficulté du problème SAT.

1 Introduction

Le problème de la satisfiabilité (SAT), étant un des premiers problèmes NP-complets, est important aussi bien en théorie qu'en pratique. Du point de vue théorique, il joue un rôle central dans les calculs de complexités. Du point de vue pratique, il se traduit par beaucoup de nombreux problémes concrets tels que la vérification formelle, les ressources d'allocation et la gestion de l'emploi du temps qui peuvent être naturellement codés en problèmes SAT.

Depuis les années 1990, un phénomène de transition de phase, en probabilité de l'existence d'une solution, fut observé pour beaucoup de problèmes en informatique. Cette transition coïncide avec la région où la difficulté des instances s'accroît, où SAT est un des premiers problèmes susceptibles de mettre en évidence un tel phénomène.

Dans l'étude de transitions de phases pour les problèmes NP-complets, ce sont les problèmes 3-SAT aléatoires qui ont retenu le plus d'attention depuis une dizaine d'années. Beaucoup de travaux ont été réalisés pour prouver que les bornes inférieures et supérieures au voisinage des points de seuil, et les meilleures bornes inférieure et supérieure actuelles pour 3-SAT aléatoire sont respectivement de 3,52 [9] et 4,506 [2] (la valeur exacte n'étant pas encore fixée à ce jour). Dans la poursuite de la conception d'algorithmes rapides pour des problèmes NP-complets, et pour évaluer expérimentalement la performance asymptotique de ces algorithmes, il est nécessaire de générer des instances de tailles croissantes dont la difficulté peut être garantie. Principalement grâce à leur simplicité et leur difficulté, les problèmes 3-SAT aléatoires sont utilisés comme une importante source pour générer des benchmarks à soumettre aux différents algorithmes ou solveurs (ils représentent une des trois principales catégories de benchmarks adoptées par la *SAT Competition*[1]).

En fait, 3-SAT aléatoire a grandement motivé le développement d'algorithmes tels que satz [11] et adaptg2wsat

*nouredine.ould@u-picardie.fr

[1]http ://www.satcompetition.org

[12] au cours des dernières années. Toutefois, l'absence de structure dans les instances 3-SAT aléatoires fait qu'elles sont tout à fait différentes de celles générées par un codage SAT de problèmes du monde réel, de sorte qu'un algorithme efficace pour 3-SAT aléatoire ne l'est pas nécessairement pour les problèmes SAT issus du monde réel. Pour fournir une bonne alternative aux critères purement aléatoires, le problème de quasigroup (*latine Squares* ou QCP) (voir par exemple [10]) a été présenté comme un domaine de benchmarks structurés [1, 6], ce qui a grandement fait progresser l'étude du problème SAT, surtout empiriquement. Cependant, par rapport au 3-SAT aléatoire, ce domaine n'est pas encore assez simple à comprendre et à utiliser, et le codage SAT de ce problème n'est pas encore assez affiné pour l'étude asymptotique du comportement d'un algorithme. Par exemple, le codage d'une instance SAT $\mathcal{F}_1$ du *latine Squares* d'ordre 20 utilise naturellement 8000 (20^3) variables booléennes, alors qu'un codage d'une instance SAT $\mathcal{F}_2$ du *latine Squares* d'ordre 21 fait naturel usage de 9261 (21^3) variables booléennes. Ceci est différent de 3-SAT aléatoire pour lequel on peut continuellement augmenter la taille de l'instance par l'ajout de 50 variables pour étudier asymptotiquement le comportement d'un algorithme.

Il est donc utile de fournir un effort de plus afin de trouver un nouveau modèle SAT avec de meilleures fonctionnalités pour permettre de mieux étudier le problème SAT, tout en étant presque aussi simple que le modèle 3-SAT aléatoire. Idéalement, un algorithme efficace pour ce genre de benchmarks SAT devrait être également efficace pour une large palette de problèmes SAT aléatoires et structurés.

Le Problème de Satisfaction de Contraintes[2] (CSP) est une généralisation du problème SAT. Le CSP peut être transformé en un problème SAT en utilisant des codages différents, tels que le codage direct, le codage de support et le codage logarithmique. Dans [20], un modèle de CSP appelé Modèle RB a été proposé. Contrairement au 3-SAT aléatoire, le Modèle RB a des points de seuil qui peuvent être précisément situés. En outre, à l'heure actuelle de nombreuses études sur des algorithmes incomplets s'appuient beaucoup sur des expérimentations et analyses empiriques pour des instances satisfiables mais assez difficiles. Des travaux antérieurs [19] montrent que le Modèle RB peut également être utilisé pour générer des instances difficiles et satisfiables en utilisant une stratégie très simple. Puisque le Modèle RB a les avantages mentionnés ci-dessus, il serait donc pertinent d'explorer si ces propriétés intéressantes se retrouvent lorsque l'on traduit en instances SAT des instances générées par les modèles RB (les différents codages pourraient en effet avoir des propriétés très différentes). Dans ce papier, nous effectuons une étude plus détaillée et systématique utilisant trois méthodes de codage, à la recherche des différents paramètres de contrôle, de la transition de phase en satisfiabilité, du rapport de backbone ainsi que de la difficulté au voisinage du seuil. Selon nos connaissances, c'est la première fois qu'une étude est réalisée par rapport aux comportements respectifs de ces trois méthodes de codage (direct, de support et logarithmique) en utilisant des instances asymptotiquement dures. En outre, nous proposons dans ce papier un modèle simple et aléatoire issu des fonctionnalités du Modèle RB, et nous espérons que ce nouveau modèle sera utilisable par la suite pour l'étude des problèmes SAT.

Le reste de ce papier est organisé comme suit. Tout d'abord dans la section 2, nous introduirons quelques définitions de base ainsi que le Modèle RB. Ensuite, un aperçu des codages CSP en SAT est fait dans la section 3. Par la suite, nous présenterons nos résultats expérimentaux ainsi que l'analyse dans la section 4. Après cela, nous proposons un nouveau modèle SAT aléatoire appelé RB-SAT, et précisons ses avantages par rapport au 3-SAT aléatoire à la section 5. Enfin, dans la section 6 nous ferons nos conclusions et discuterons des travaux futurs.

2 Préliminaires

2.1 SAT et k-SAT aléatoire

Une instance SAT est une formule propositionnelle sous Forme Normale Conjonctive CNF (en anglais : Conjonctive Normal Form). Le problème SAT consiste à déterminer si une telle formule est satisfiable ou non. En termes plus formels, cela se résume dans les définitions suivantes :

Variables booléennes : Variables pouvant avoir pour valeur l'une des valeurs de vérité *vrai* ou *faux*.

Littéraux : Variables (littéraux positifs, par exemple P) ou leurs négations (littéraux négatifs, par exemple $\neg P$).

Clauses : Disjonction de littéraux, par exemple $P \vee \neg Q$.

Formule CNF : Conjonction de clauses, par exemple $(P \vee \neg Q) \wedge (P \vee Q)$.

Problème SAT : Étant donnée une formule CNF F, demander s'il existe une affectation de valeurs de vérité à des variables booléennes telles que toutes les clauses de F soient satisfaites (évaluées à *vrai*). Si tel est le cas, la formule F est dite satisfiable, sinon elle est dite insatisfiable.

[2] Constraint Satisfaction Problem

k-**SAT** : C'est le problème SAT restreint à des clauses avec exactement *k* littéraux (des clauses ayant toutes une longueur exactement égale à *k*). Il est assez connu que le problème *k*-SAT est un problème NP-complet pour $k \geq 3$. Pour beaucoup de problèmes NP-complets, la probabilité de générer aléatoirement des instances satisfiables montre une forte transition de phase quand un paramètre de contrôle est varié, à partir d'une région où la probabilité est près de 1 à une région où la probabilité est près de 0. La transition de phase correspond en général à un pic de coût de recherche pour les différents algorithmes les plus connus de la littérature, c'est-à-dire que les instances générées grâce aux paramètres de contrôle près de la valeur critique sont les plus difficiles à résoudre.

Le problème *k*-SAT aléatoire est une distribution probabiliste des instances *k*-SAT. Une instance *k*-SAT aléatoire avec *n* variables et *m* clauses est construite par la sélection de *m* clauses de manière uniforme et indépendante de l'ensemble de toutes les *k*-clauses possibles sur *n* variables, où chaque clause a exactement *k* littéraux sans redondance (sans répétition d'occurrence) de variables dans une même clause. Pour *k*-SAT aléatoire, le ratio des clauses par rapport aux variables (c'est-à-dire m/n) agit souvent comme le paramètre de contrôle.

2.2 Le problème CSP

Un Problème de Satisfaction de Contraintes (CSP) est généralement défini comme un triplet $\langle X,D,C\rangle$, où $X=\{x_1,...,x_n\}$ est un ensemble de variables, D est un domaine de valeurs et $C=\{C_1,...,C_p\}$ est un ensemble de contraintes. Chaque contrainte $C_i=\langle S_i,R_i\rangle$ concerne un ensemble de variables $S_i=\{x_{i_1},...,x_{i_k}\}$ (où k est l'arité de C_i) et a une relation associée R_i qui spécifie les tuplets de valeurs compatibles pour ces variables dans S_i.

Une contrainte est dite satisfaite si le tuplet des valeurs assigné aux variables dans cette contrainte est compatible. Une solution au CSP est une affectation de valeurs de vérité à toutes les variables telle que toute contrainte soit satisfaite. Un CSP ayant au moins une solution est dit satisfiable (sinon, insatisfiable). Le but du CSP est de trouver une solution, s'il est satisfiable, ou de prouver qu'il est insatisfiable. En général, le problème CSP est NP-complet.

2.3 Le Modèle RB

Modèle RB [20] : Une classe d'instances CSP aléatoires, générées suivant le Modèle RB qui s'exprime par $RB(k,n,\alpha,r,p)$, où pour toute instance :

- $k \geq 2$ est l'arité de chaque contrainte,
- $n \geq 2$ est le nombre de variables,
- $\alpha > 0$ détermine la taille du domaine $d=n^\alpha$ de chaque variable,
- $r > 0$ détermine le nombre $m=r.n.ln(n)$ de contraintes,
- $0 < p < 1$ détermine le nombre $t=pd^k$ de tuplets rejetés (incompatibles ou exclus) dans chaque relation.

On suppose que tous les domaines des variables contiennent le même nombre de valeurs $d=n^\alpha$ dans le Modèle RB. Dans ce papier, nous limitons notre attention au cas binaire du Modèle RB, c'est-à-dire le cas où k=2. La génération aléatoire des instances CSP dans le Modèle RB est effectuée selon les deux étapes suivantes :

Étape 1. Sélectionner avec répétition $m=rnln(n)$ contraintes aléatoires. Chaque contrainte aléatoire est formée par sélection sans répétition de k variables parmi les n variables du problème.

Étape 2. Pour chaque contrainte, sélectionner uniformément et sans répétition $t=pd^k$ tuplets incompatibles de valeurs.

L'instance peut être forcée à être satisfiable par l'affectation aléatoire d'une valeur à chaque variable, ensuite garder chaque contrainte satisfaite lors de la sélection des tuplets incompatibles de valeurs pour cette contrainte dans l'*Étape 2*.

L'existence de transitions de phase dans le Modèle RB fut prouvée dans [20] et les points de seuil exacts y sont donnés. Plus précisément, nous avons les théorèmes suivants : (où Pr(Sat) exprime la probabilité qu'une instance aléatoire $P \in RB(k,n,\alpha,r,p)$ soit satisfiable) :

Theorem 2.1 *Si k,* $\alpha > \frac{1}{k}$ *et* $p \leq \frac{k-1}{k}$ *sont des constantes alors*

$$\lim_{n\to\infty} \Pr(Sat) = \begin{cases} 1 & if\ r < r_{cr} \\ 0 & if\ r > r_{cr} \end{cases}$$

où $r_{cr} = -\frac{\alpha}{\ln(1-p)}$.

Theorem 2.2 *Si k,* $\alpha > \frac{1}{k}$ *et* $p_{cr} \leq \frac{k-1}{k}$ *sont des constantes alors*

$$\lim_{n\to\infty} \Pr(Sat) = \begin{cases} 1 & if\ p < p_{cr} \\ 0 & if\ p > p_{cr} \end{cases}$$

où $p_{cr} = 1 - e^{-\frac{\alpha}{r}}$.

3 Codages de CSP en SAT

3.1 Codage direct

Dans [18] Walsh avait introduit le codage direct qui est considéré comme étant, à la fois, le codage le plus naturel et le plus largement utilisé. Plus précisément, une variable SAT x_{v_i} est vraie si et seulement si la valeur de vérité i est affectée à la variable CSP v (i et j étant

dans $dom(v)$=$\{0,...,d-1\}$). Le codage direct est constitué de trois types de clauses. Les « au-moins-une » clauses $x_{v_0} \vee x_{v_2} \vee ... \vee x_{v_{d-1}}$ qui expriment le fait que chaque variable CSP doit prendre au moins une valeur du domaine, les « au-plus-une » clauses $\overline{x}_{v_i} \vee \overline{x}_{v_j}$ signifiant que toute variable CSP peut prendre au plus une valeur du domaine et les clauses conflictuelles (ou de « conflit ») $\overline{x}_{v_i} \vee \overline{x}_{w_j}$ qui sont, quand à elles, utilisées afin d'exprimer les conflits.

Par exemple, on considère le problème CSP suivant : $A > B$, avec A et B appartenant à l'ensemble de valeurs $\{1,2,3\}$. Le codage direct est exprimé comme suit :
au-moins-une : $a_1 \vee a_2 \vee a_3, b_1 \vee b_2 \vee b_3$
au-plus-une : $\neg a_1 \vee \neg a_2, \neg a_1 \vee \neg a_3, \neg a_2 \vee \neg a_3, \neg b_1 \vee \neg b_2, \neg b_1 \vee \neg b_3, \neg b_2 \vee \neg b_3$
conflit : $\neg a_1 \vee \neg b_1, \neg a_1 \vee \neg b_2, \neg a_1 \vee \neg b_3, \neg a_2 \vee \neg b_2, \neg a_2 \vee \neg b_3, \neg a_3 \vee \neg b_3$

3.2 Codage de support

Similaire au codage direct, le codage de support génère les clauses « au-moins-une » et « au-plus-une » comme le codage direct. En revanche, ce codage remplace les clauses de « conflit » du codage précédent par les clauses de support définies comme suit. Si $i_1 ... i_k$ sont des valeurs de support dans le domaine de la variable CSP v pour la valeur j dans le domaine de la variable CSP w, alors ajouter la clause de support (ou de soutien) $x_{v_{i_1}} \vee ... \vee x_{v_{i_k}} \vee \overline{x}_{v_{w_j}}$.

Il est donc évident que le codage direct et celui du support semblent avoir beaucoup de similitudes. En effet, il fut montré par Gent [5] que les clauses de support peuvent être dérivées à partir du codage direct. Ce qui suit montre le codage du support correspondant à l'exemple précédent.
au-moins-une : $a_1 \vee a_2 \vee a_3, b_1 \vee b_2 \vee b_3$
au-plus-une : $\neg a_1 \vee \neg a_2, \neg a_1 \vee \neg a_3, \neg a_2 \vee \neg a_3, \neg b_1 \vee \neg b_2, \neg b_1 \vee \neg b_3, \neg b_2 \vee \neg b_3$
support : $\neg a_1, \neg a_2 \vee b_1, \neg a_3 \vee b_1 \vee b_2, \neg b_1 \vee a_2 \vee a_3, \neg b_2 \vee a_3, \neg b_3$

3.3 Le codage logarithmique

Dans le codage logarithmique, nous utilisons m=$\lceil \log_2 d \rceil$ variables logiques pour représenter les domaines et chacune des 2^m combinaisons représente une affectation possible. Le codage logarithmique utilise un nombre logarithmique au lieu d'un nombre linéaire de variables booléennes pour coder les domaines. Ce qui permet de générer des modèles de taille plus petite. Pour chaque valeur de bit CSP variable/domaine, il existe une variable SAT correspondante, et chaque conflit a une clause SAT qui lui correspond également. Les clauses « au-moins-une » et « au-plus-une » sont inutiles pour le codage logarithmique. Au lieu de cela, ce codage dispose des clauses dites à « valeur interdite » [15] qui sont nécessaires pour exclure les valeurs en trop quand la cardinalité des domaines n'est pas une puissance de deux. Dans l'exemple utilisé, le codage logarithmique est montré comme suit :
valeur-interdite : $a_1 \vee a_0, b_1 \vee b_0$
conflit : $a_1 \vee \neg a_0 \vee b_1 \vee \neg b_0, a_1 \vee \neg a_0 \vee \neg b_1 \vee b_0, a_1 \vee \neg a_0 \vee \neg b_1 \vee \neg b_0, \neg a_1 \vee a_0 \vee \neg b_1 \vee b_0, \neg a_1 \vee a_0 \vee \neg b_1 \vee \neg b_0, \neg a_1 \vee \neg a_0 \vee \neg b_1 \vee \neg b_0$

4 Expérimentations

Il est bien connu que les codages SAT d'un problème peuvent avoir des propriétés très différentes : certains facteurs de la difficulté peuvent être supprimés, mais certaines difficultés de résolution peuvent être introduites ou ajoutées. Par exemple, lorsque le problème de parité DIMACS, intrinsèquement facile, a été codée en un problème SAT *challenge* pour les solveurs SAT [16], il fut observé dans [1] que le problème des quasigroups devient plus facile à résoudre par satz après l'avoir codé en SAT. Les caractéristiques des codages qui peuvent être efficacement résolus représente aussi un *challenge* donné dans [16].

Dans cette section, nous donnons quelques résultats expérimentaux représentatifs des différents codages. Quelques analyses théoriques sont déjà faites dans [21], utilisant le codage direct et concernant les instances CSP et des résultats empiriques sont présentés dans [19]. Afin de réaliser une meilleure comparaison et d'élargir le spectre de l'applicabilité des résultats précédents, nous réalisons des expérimentations comparatives et modifions la plupart des paramètres de contrôle en plus de ceux du [19].

Ces expérimentations ont été réalisées sur un *PC Pentium IV 3 GHz 1GMb* sous *Windows XP Professional SP2*. Tous les points de données dans les figures, de 1 à 8, sont obtenus en utilisant 50 instances. Les algorithmes (solveurs) utilisés dans ces expérimentations sont bien connus dans la littérature, incluant à la fois des algorithmes complets (satz [11] et zchaff [14]) et d'autres incomplets (walksat [17] et adaptnovelty [8]).

4.1 Coût de recherche des codages SAT pour le Modèle RB

Il a été montré dans [19] que les plus difficiles instances CSP sont situées assez près du seuil théorique, et que les instances forcées et les instances non forcées ont tendance à y avoir des difficultés similaires. Afin de voir si ces résultats sont toujours valables pour les instances SAT obtenues en utilisant les différents codages, nous réalisons des expérimentations comparatives en utilisant les trois codages présentés ci-dessus (direct, de support et logarithmique).

Dans les figures 1 et 2 (gauche), nous avons étudié la difficulté de résoudre avec zchaff les instances SAT du Modèle RB générées au voisinage du seuil théorique $p_{cr} \approx 0.23$ donné par le théorème 2.2 pour k=2, α=0.8,

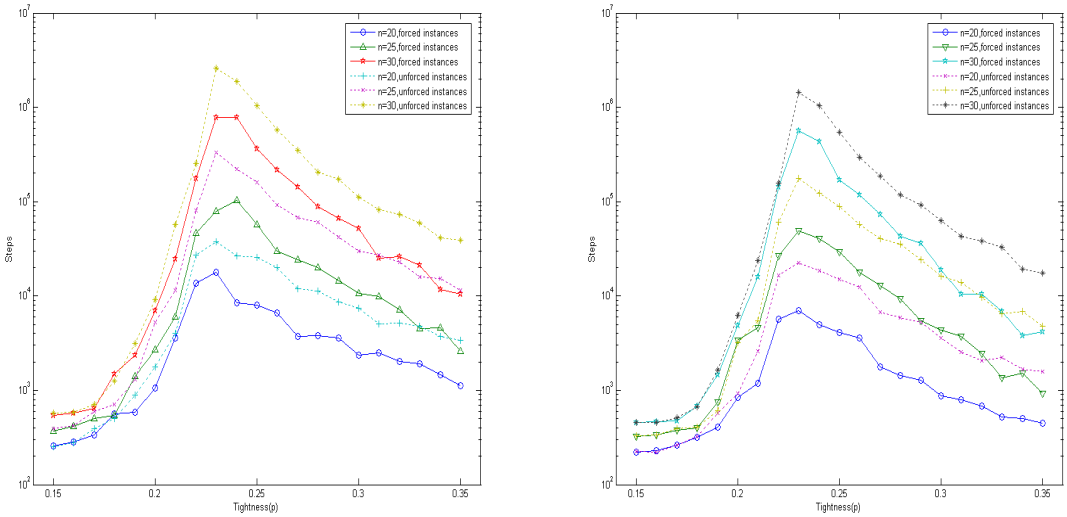


FIG. 1 – Moyenne du coût de recherche d'une solution pour des instances codées en $RB(2,\{20,25,30\},0.8,3,p)$ avec zchaff, utilisant le codage direct (à gauche) et le codage de support (à droite)

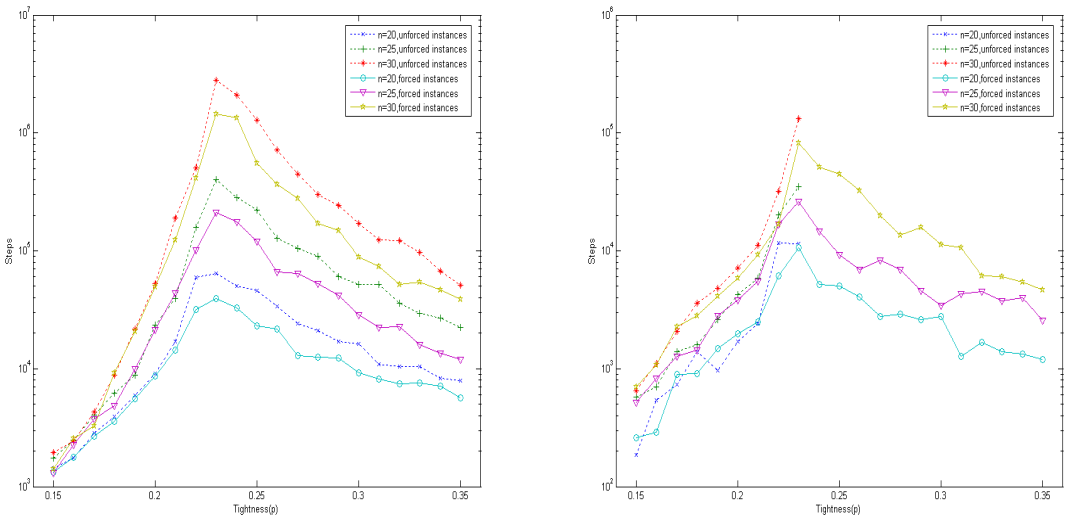


FIG. 2 – Moyenne du coût de recherche d'une solution pour des instances codées en $RB(2,\{20,25,30\},0.8,3,p)$ avec zchaff, utilisant le codage logarithmique (à gauche) et adaptnovelty utilisant le codage direct (à droite)

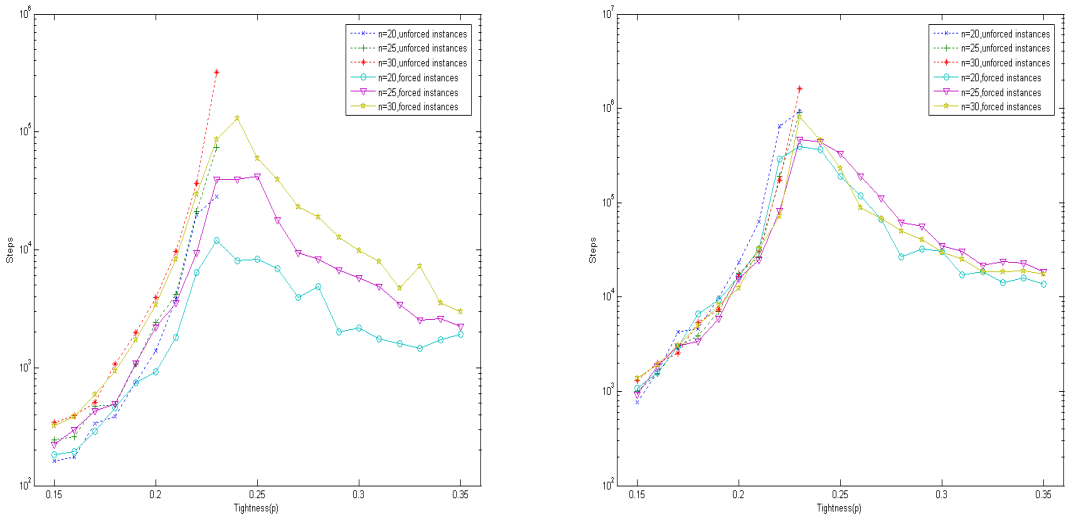


FIG. 3 – Moyenne du coût de recherche d'une solution pour des instances codées en $RB(2,\{20,25,30\},0.8,3,p)$ avec adaptnovelty, utilisant le codage du support (à gauche) et le codage logarithmique (à droite)

r=3 et $n \in \{20,30,40\}$.

Ces figures montrent clairement que quelque soit la méthode de codage utilisée, la plupart des instances les plus difficiles apparaissent toujours près du point du seuil théorique et il y a peu de différence, en difficulté, entre les instances forcées et celles qui ne le sont pas. Une telle difficulté augmente de façon exponentielle selon n (utilisation d'une échelle logarithmique). Dans les figures 2 (droite) et 3, nous utilisons le solveur incomplet adaptnovelty à la place de zchaff pour résoudre les instances, et il peut être constaté que les résultats sont sensiblement identiques. Pour le codage direct (que nous utiliserons pour définir le nouveau modèle SAT plus loin dans la section 5), nous avons plus d'expérimentations avec les deux algorithmes bien connus tels que satz et walksat, et les résultats sont présentés dans la figure 4.

Nous avons également effectué des expérimentations pour tester la satisfiabilité des instances non forcées. Leurs résultats sont montrés dans la figure 5. Comme attendu, au seuil théorique $p_{cr} \approx 0.23$ pour $RB(2,\{20,25,30\},0.8,3,p)$, nous observons la plus nette (aigüe) transition de phase en satisfiabilité. Il peut être constaté que la largeur de la région contenant la transition de phase se rétrécit (diminue) lorsque n est assez grand, ce qui est très similaire à ce qui a été trouvé pour le k-SAT aléatoire.

4.2 Transition de phase dans le backbone des instances forcées

Pour les instances forcées, nous étudions le rapport backbone dans nos expérimentations. La notion du backbone d'un problème SAT fut introduite dans [13] pour désigner le rapport des variables qui prennent les mêmes valeurs dans toutes les solutions. Le rapport backbone (ratio des variables du backbone au nombre total de variables) est une propriété des problèmes CSP et SAT qui est bien définie pour les distributions satisfiables. Le phénomène de transition de phase dans le rapport du backbone a été observé dans [1].

Dans nos expérimentations, nous avons constaté la présence de ce phénomène quelque soit la méthode de codage adoptée.
Les figures 6 et 7 (gauche) montrent le rapport du backbone comme une fonction d'étroitesse (d'exiguïté) p des différents nombres de variables n. Une forte transition de phase apparaît dans le rapport du backbone, qui correspond au pic du coût de difficulté au cours de la recherche montrée ci-dessus (où p=0.23). On peut également observer que plus n croît plus la largeur de la région de transition de phase diminue.

De plus, nous avons trouvé que les sous figures (gauche et doite) de la figure 6 sont presque identiques, et il semble que le codage direct et le codage de support ont effectivement des points fondementaux en commun. Cet intriguant phénomène mérite une recherche plus approfondie.

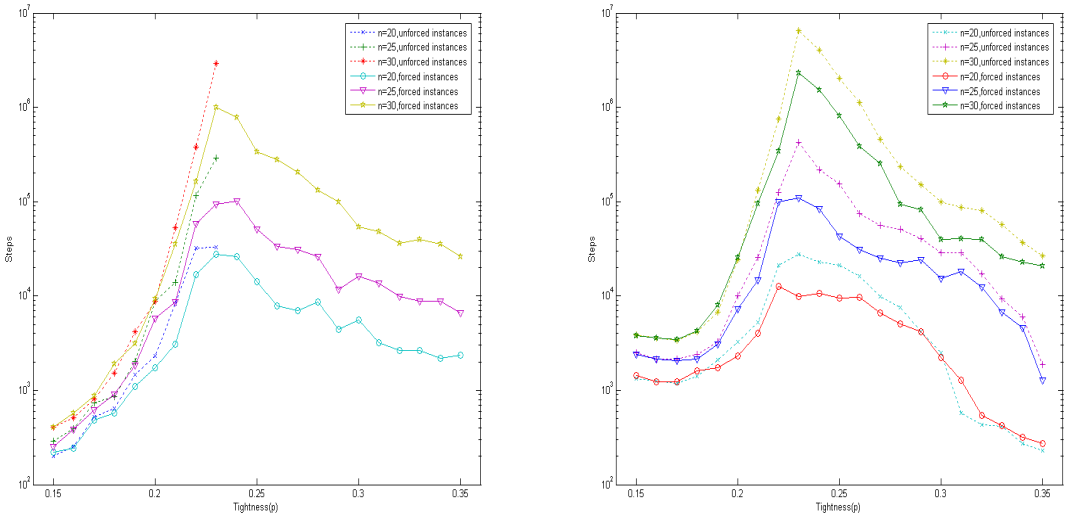


FIG. 4 – Moyenne du coût de recherche d'une solution pour des instances codées en $RB(2,\{20,25,30\},0.8,3,p)$ avec walksat (à gauche) et satz (à droite), utilisant le codage direct

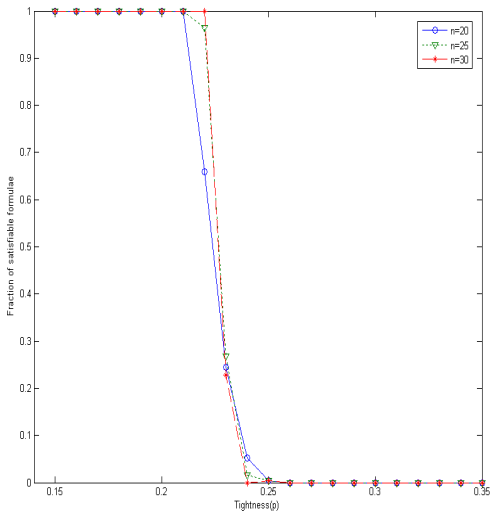


FIG. 5 – Transition de phase en satisfiabilité pour $RB(2,\{20,25,30\},0.8,3,p)$

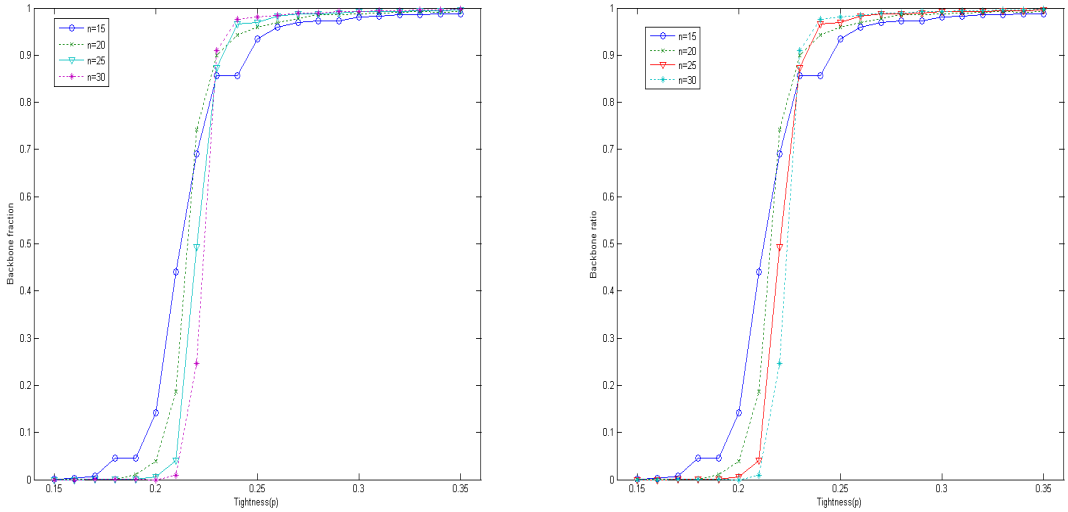


FIG. 6 – Transition de phase dans le rapport du backbone pour les instances codées en $RB(2,\{15,20,25,30\},0.8,3,p)$, utilisant le codage direct (à gauche) et le codage de support (à droite)

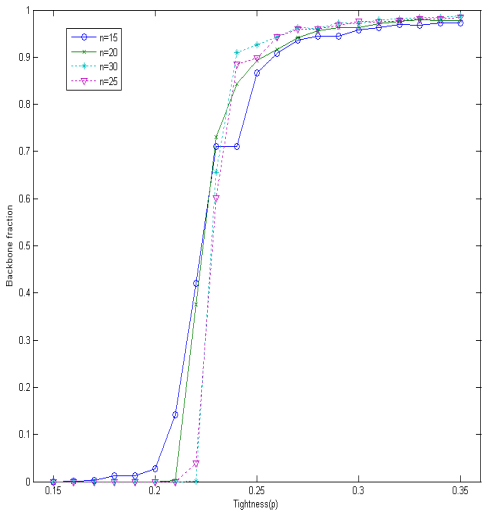


FIG. 7 – Transition de phase dans le rapport du backbone pour les instances codées en $RB(2,\{15,20,25,30\},0.8,3,p)$, utilisant le codage algorithmique

4.3 Difficulté des instances à proximité du seuil

Les figures 1 à 5 ont montré que la difficulté des instances forcées est tout à fait similaire à celle des instances non forcées, laquelle s'accentue exponentiellement lorsque n augmente. Pour confirmer et illustrer l'étendu du spectre de l'applicabilité de ce résultat, nous nous sommes concentrés sur un point juste en dessous du seuil. Pour cela, différentes valeurs de paramètres α et r ont été choisies. En outre, nous voulons aussi étudier au préalable la façon dont les coûts de recherche diffèrent lorsque les différents codages sont utilisés.

La figure 8 présente le coût de recherche de solution avec zchaff, à la fois, pour les instances SAT forcées et les non forcées générées grâce au Modèle RB avec $p_{cr} - 0.01 \approx 0.40$ pour k=2, α=0.8, r=1.5 et $n \in [20...40]$, utilisant les trois différents codages. Une fois de plus, le résultat confirme ce qui a été observé auparavant ; à savoir que les deux types d'instances (forcées et non forcées) ont une complexité exponentielle très similaire à proximité du seuil.

La différence entrainée par les codages montre que, généralement, le codage logarithmique produit des instances difficiles, tandis que le codage de support en génère des plus faciles. Ceci concorde bien avec les comparaisons expérimentales faites dans [4, 5, 7] (ces comparaisons avaient été réalisées par des chercheurs différents, utilisant des méthodologies tout à fait différentes). Avec un examen plus attentif, on peut voir que, bien que ces lignes commencent à partir de différents points, elles sont presque parallèles lorsque n augmente suffisamment.

On peut donc conclure que la différence en performance entre les codages reste relativement identique, même quand la taille du problème grandit (ce qui génère des instances difficiles). Cela concorde bien avec ce qui a été trouvé auparavant, et d'après nos connaissances, c'est la première fois que le comportement asymptotique des trois codages est étudié à l'aide des instances dures.

5 Un nouveau modèle SAT aléatoire

Les avantages du Modèle RB résident dans sa simplicité, la difficulté des instances qu'il produit et ses bonnes propriétés théoriques. Les résultats expérimentaux de la section 4 montrent que quelque soit la méthode de codage utilisée, les instances SAT obtenues grâce au Modèle RB héritent de bonnes caractéristiques de RB : aisance d'utilisation, précision en transition de phase et difficulté de résolution. Nous pouvons alors définir un modèle SAT simple correspondant à chacune des trois méthodes de codage. Le modèle suivant correspond au codage direct du Modèle RB (avec k=2) :

Étape 1. *Générer n ensembles disjoints de variables booléennes dont chacun a pour cardinalité* n^{α} *(où* $\alpha > 0$ *est une constante). Pour tout ensemble, générer une clause qui est la disjonction de toutes les variables de cet ensemble et pour toutes deux variables x et y dans ce même ensemble, générer une clause binaire* $\neg x \vee \neg y$.

Étape 2. *Sélectionner aléatoirement deux ensembles disjoints et générer sans répétitions* $pn^{2\alpha}$ *clauses de la forme* $\neg x \vee \neg z$ *où x et z sont deux variables sélectionnées aléatoirement et respectivement à partir des deux ensembles (où* $0 < p < 1$ *est une constante) ;*

Étape 3. *Faire l'Étape 2 (avec répétitions)* $rn\ln n$-1 *fois (où* $r > 0$ *est une constante).* Ce modèle, que nous nommons RB-SAT(n, α, r, p) (puisqu'il est basé sur le Modèle RB), est évidemment très facile à comprendre, même sans aucune connaissance du CSP ou de l'encodage, et peut donc être utilisé très convenablement dans les études théoriques et expérimentales du problème SAT. Nous pensons que RB-SAT fournit un autre bon modèle SAT aléatoire en plus du k-SAT aléatoire. À l'instar du k-SAT, il a été prouvé dans [21] que le codage SAT direct des formules de type RB n'ont presque certainement pas d'arbres de résolution de preuves en-dessous de l'exponentielle, ce qui implique que RB-SAT est difficile pour les résolutions d'arbres. Les résultats expérimentaux de la section 4 confirment que les instances RB-SAT sont difficiles. Étant donné que le codage de CSP en SAT n'a pas d'impact sur la satisfiabilité, nous savons que les transitions de phase exactes montrées dans [20] existent également dans RB-SAT, c'est à dire que les théorèmes 2.1 et 2.2 s'appliquent facilement à RB-SAT(n, α, r, p) pour donner la valeur exacte du seuil correspondant à la transition de phase de satisfiabilité, ce qui est très utile pour l'étude de la difficulté des problèmes SAT. On note que la valeur exacte du seuil correspondant à la transition de phase de la satisfiabilité de k-SAT demeure inconnue à ce jour.

Les instances de RB-SAT(n, α, r, p) contiennent $n^{1+\alpha}$ variables et $O(n^{1+2\alpha}\ln(n))$ (c'est à dire, $n(n^{\alpha}$-1)/2+1)+$rn\ln(n)pn^{2\alpha}$) clauses. En pratique, nous utilisons $0.5 < \alpha < 1$ pour générer des instances difficiles au seuil selon les théorèmes 2.1 et 2.2. Ainsi, le nombre de variables booléennes augmente relativement lentement lorsque n augmente.

En outre, les instances RB-SAT peuvent être facilement générées en utilisant la même stratégie que RB. Ces instances sont assurément satisfiables et ayant une difficulté similaire à celle des instances non forcées. En plus, un phénomène de transition de phase dans le rapport de backbone de ces instances est observé, caractérisant ainsi leur difficulté. Les pics de difficulté des instances forcées

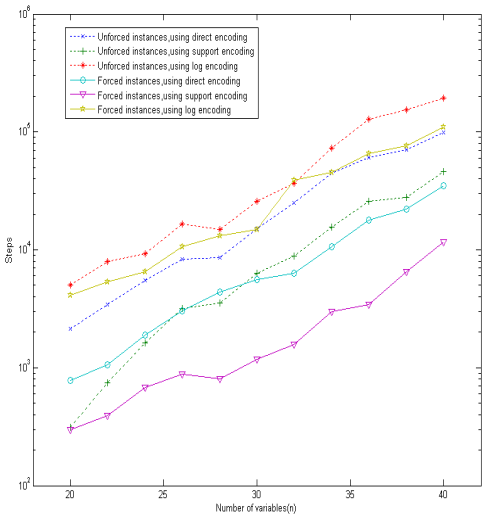


FIG. 8 – Coût moyen de recherche de solution pour des instances codées en $RB(2,\{20...40\},0.8,1.5,p_{cr}-0.01)$ avec zchaff

et celui des instances non forcées coïncident.

La figure 9 (gauche) compare 3-SAT aléatoire avec un ratio clause/variable de 4.25 et RB-SAT($n,0.8,3,0.23$) utilisant trois solveurs de l'état-de-l'art : minisat [3], satz et adaptg2wsat+ [12]. Notons que les instances 3-SAT aléatoires sont au seuil empirique lorsque les instances RB-SAT sont au seuil donné dans le théorème 2.2. Les instances comparées de 3-SAT et RB-SAT ont le même nombre de variables, correspondant à n=20, 23, 25, 28, 30, 32, 35, 38, et 40 dans RB-SAT. La figure 9 (droite) compare les instances non forcées mais satisfiables de RB-SAT et les instances forcées de RB-SAT avec les même paramètres, après utilisation de satz ou minisat pour filtrer les instances insatisfiables non forcées de RB-SAT.

Minisat est choisi pour sa haute performance dans les récentes *SAT competition*. Nous en utilisons la récente version 2.8. Quant au solveur satz, il est choisi pour sa haute performance à la fois pour les instances 3-SAT aléatoires et pour les problèmes structurés tels que celui des quasi-groups. Le solveur adaptg2wsat+ est choisi puisqu'il résout le plus grand nombre d'instances entre tous les solveurs basés sur les algorithmes de la Recherche Locale dans *SAT competition* de 2007 [3]. À chaque point, où 250 instances 3-SAT aléatoires et 250 instances RB-SAT sont résolues, la moyenne des temps d'exécution est affichée.

Sur la figure 9, nous observons que :

– Comme pour 3-SAT aléatoire, on peut sans cesse augmenter la taille des instances RB-SAT tout en les maintenant solubles. La palette de taille des instances RB-SAT solubles est encore plus grande que celle du 3-SAT aléatoire, car les instances RB-SAT difficiles de 750 variables restent solubles, tandis que les instances 3-SAT aléatoires difficiles peuvent difficilement dépasser 500 variables. En outre, il existe de nombreux exemples de la même taille, ce qui est très utile pour évaluer le comportement asymptotique d'un algorithme.
– Satz est plus rapide que minisat pour 3-SAT aléatoire, alors que minisat est plus rapide que satz pour RB-SAT. Ceci est dû à l'absence de structure dans 3-SAT aléatoire, tandis qu'une instance RB-SAT peut avoir quelques structures, par exemple pour chaque ensemble disjoint de variables booléennes, il y a des clauses imposant qu'une et une seule variable dans cet ensemble peut prendre la valeur 1. En d'autres termes, afin de résoudre efficacement les instances RB-SAT, un solveur devrait faire mieux que minisat pour la partie aléatoire de RB-SAT et être plus performant que satz pour traiter les dépendances des variables dans RB-SAT, motivant ainsi le développement de solveurs efficaces à la fois pour les problèmes structurés et les problèmes aléatoires.
– Pour le solveur adaptg2wsat+, les instances RB-SAT sont plus difficiles que les 3-SAT aléatoires, ce qui signifie que la recherche locale devrait être améliorée afin de traiter de manière plus efficace les dépendances des variables. En effet, ceci est un des dix *challenges* donnés dans [16]. RB-SAT peut donc être un bon moyen pour répondre à ce défi.

[3] www.satcompetition.org

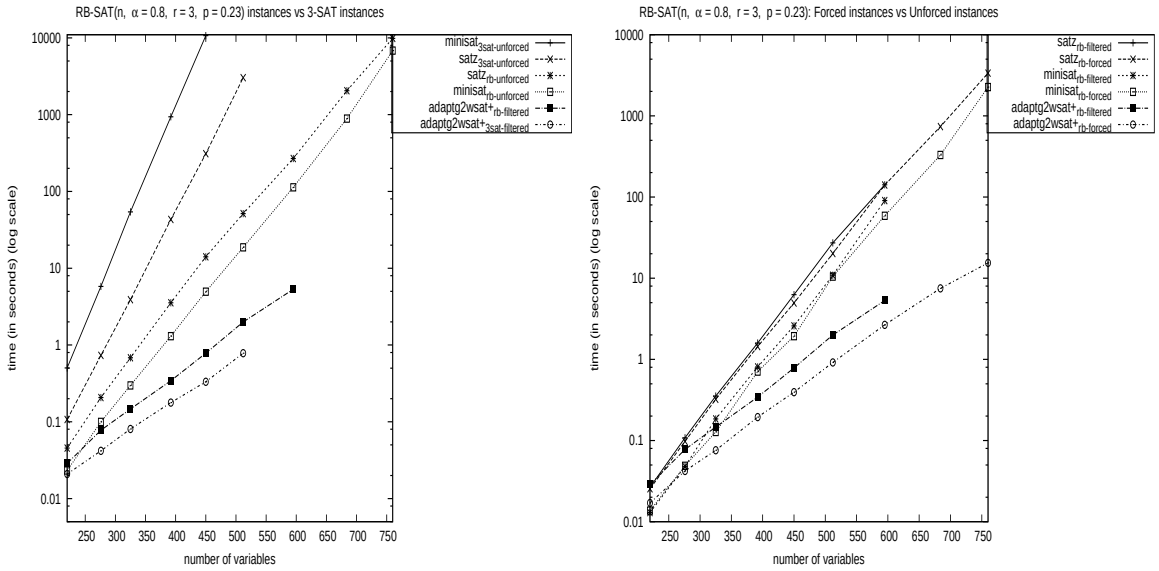


FIG. 9 – Comparaison de 3-SAT aléatoire avec RB-SAT (à gauche), et des instances non forcées RB-SAT avec des instances forcées RB-SAT (à droite), en terme de moyenne des temps d'exécution en seconde des 250 instances résolues par solveur à chaque point. Ces expérimentation ont été réalisées sur un mac pro 2.8 Ghz avec 2x4 processeurs.

En effet, puisqu'il n'est pas très facile d'augmenter le nombre de variables pour de nombreux autres problèmes SAT structurés tout en gardant ces problèmes solubles, on peut assez facilement générer beaucoup d'instances difficiles et satisfiables de RB-SAT tout en augmentant soigneusement leurs tailles (en les maintenant toujours satisfiables) pour mieux évaluer le comportement asymptotique d'une approche de recherche locale, lors du traitement des dépendances des variables. De plus, ces instances sont également aléatoires, permettant ainsi à l'approche développée de sauvegarder son efficacité pour les instances aléatoires. Observons que même si les instances RB-SAT forcées sont plus faciles que les non forcées satisfiables pour adaptg2wsat+, la différence en performance de adaptg2wsat+ semble être la même quand les instances deviennent assez grandes.

– RB-SAT est plus similaire aux codages des formules SAT relevant des problèmes issus du monde réel que 3-SAT aléatoire. Ainsi, un algorithme efficace pour RB-SAT devrait être probablement plus efficace pour les problèmes SAT du monde réel qu'un algorithme efficace uniquement pour 3-SAT aléatoire.

Dans la pratique, on peut modifier légèrement l'étape 3 pour choisir les deux différents ensembles disjoints sans répétition, les instances RB-SAT générées deviennent alors nettement plus difficiles, ce qui devrait être encore plus utile pour une meilleure mise au point des solveurs SAT. On admet que les théorèmes 2.1 et 2.2 sont toujours valables avec une telle modification car, pour des n très grands, il y a très peu de différence avec ou sans répétition.

6 Conclusions et travaux futurs

Il est bien connu que les codages SAT pour un problème donné peuvent avoir différentes propriétés de calcul. Dans ce papier, basé sur le Modèle RB, nous faisons des analyses systématiques et détaillées de trois méthodes de codage de CSP en SAT. Il est démontré que quelque soit la méthode de codage utilisée, les instances SAT obtenues partagent les bonnes caractéristiques du Modèle RB, y compris la croissance exponentielle de la difficulté des instances qui atteint son summum à la pointe du seuil. De plus, pour les instances forcées, nous avons étudié le rapport de backbone et avons observé la transition de phase. Nous avons également comparé les comportements asymptotiques liés aux trois méthodes de codage et avons trouvé que la différence en performance de ces codages demeure relativement la même quand la taille du problème devient grande. En nous basant sur le codage direct, nous avons proposé un codage simple et naturel que nous avons appelé RB-SAT. Nous en avons ainsi présenté les avantages par rapport au 3-SAT aléatoire. Nos travaux futurs pourront inclure les analyses théoriques de RB-SAT, par exemple, la preuve de la transition de phase dans le rapport du backbone, l'analyse des moyennes du temps de la complexité pour les différents algorithmes (ou heuristiques), preuve des théorèmes 2.1 et 2.2 quand l'*Étape 1* est modifiée pour sélectionner deux ensembles disjoints de variables sans répétition et fournir (de nouvelles) ou améliorer les bornes inférieures de complexité déjà existantes. Nous utiliserons également le Modèle RB-SAT pour développer de nouvelles approches pour la résolution des problèmes SAT.

Note

Cette recherche fut particulièrement supportée par le *National 973 Program of China (Grant No. 200532CB1902)*. Une partie de ce travail a été réalisée en collaboration au laboratoire MIS à l'Université de Picardie Jules Verne.

Références

[1] D. Achlioptas, C. Gomes, H. Kautz, and B. Selman. Generating satisfiable problem instances. In *PROCEEDINGS OF THE NATIONAL CONFERENCE ON ARTIFICIAL INTELLIGENCE*, pages 256–301. Menlo Park, CA ; Cambridge, MA ; London ; AAAI Press ; MIT Press ; 1999, 2000.

[2] O. Dubois and J. Mandler. Typical random 3-SAT formulae and the satisfiability threshold. In *Proceedings of the eleventh annual ACM-SIAM symposium on Discrete algorithms*, pages 126–127. Society for Industrial and Applied Mathematics Philadelphia, PA, USA, 2000.

[3] N. Een and N. Sorensson. An extensible SAT-solver. *Proceedings of SAT-03*, pages 502–508, 2003.

[4] M.D. Ernst, T.D. Millstein, and D.S. Weld. Automatic SAT-compilation of planning problems. In *In Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence*, pages 1169–1176, 1997.

[5] I.P. Gent. Arc consistency in SAT. In *Ecai 2002 : 15th European Conference on Artificial Intelligence, July 21-26, 2002, Lyon France : Including Prestigious Applications of Intelligent Systems (PAIS 2002) : Proceedings*. IOS Press, 2002.

[6] C.P. Gomes and B. Selman. Problem structure in the presence of perturbations. In *PROCEEDINGS OF THE NATIONAL CONFERENCE ON ARTIFICIAL INTELLIGENCE*, pages 221–227, 1997.

[7] H.H. Hoos. SAT-encodings, search space structure, and local search performance. In *INTERNATIONAL JOINT CONFERENCE ON ARTIFICIAL INTELLIGENCE*, pages 296–303. LAWRENCE ERLBAUM ASSOCIATES LTD, 1999.

[8] H.H. Hoos. An adaptive noise mechanism for WalkSAT. In *Proceedings of the national conference on artificial intelligence*, pages 655–660. Menlo Park, CA ; Cambridge, MA ; London ; AAAI Press ; MIT Press ; 1999, 2002.

[9] A.C. Kaporis, L.M. Kirousis, and E.G. Lalas. The probabilistic analysis of a greedy satisfiability algorithm. *Random Structures and Algorithms*, (4) :444–480, 2006.

[10] SR Kumar. Approximating latin square extensions. *Algorithmica*, (2) :128–138, 1999.

[11] C.M. Li and A. Anbulagan. Heuristics based on unit propagation for satisfiability problems. In *Proc. of the Fifteenth International Joint Conference on Artificial Intelligence (IJCAI97)*, pages 366–371, 1997.

[12] C.M. Li, W. Wei, and H. Zhang. Combining adaptive noise and look-ahead in local search for SAT. *Lecture Notes in Computer Science*, page 121, 2007.

[13] R. Monasson, R. Zecchina, S. Kirkpatrick, B. Selman, and L. Troyansky. Determining computational complexity from characteristic 'phase transitions'. *Nature*, (6740) :133–137, 1999.

[14] MW Moskewicz, CF Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff : Engineering an efficient SAT solver. In *Design Automation Conference, 2001. Proceedings*, pages 530–535, 2001.

[15] S. Prestwich. Local search on SAT-encoded colouring problems. *Lecture Notes in Computer Science*, pages 105–119, 2004.

[16] B. Selman, H. Kautz, and D. McAllester. Ten challenges in propositional reasoning and search. In *In Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence*, 1997.

[17] B. Selman, H.A. Kautz, and B. Cohen. Noise strategies for improving local search. In *Proceedings of the national conference on artificial intelligence*, pages 337–337. JOHN WILEY & SONS LTD, 1994.

[18] T. Walsh. Sat vs csp. *Proc. CP-2000*, pages 441–456, 2000.

[19] K. Xu, F. Boussemart, F. Hemery, and C. Lecoutre. Random constraint satisfaction : Easy generation of hard (satisfiable) instances. *Artificial Intelligence*, (8-9) :514–534, 2007.

[20] K. Xu and W. Li. Exact phase transitions in random constraint satisfaction problems. *Journal of Artificial Intelligence Research*, pages 93–103, 2000.

[21] K. Xu and W. Li. Many hard examples in exact phase transitions. *Theoretical Computer Science*, (3) :291–302, 2006.

Détection de fonctions booléennes pour la preuve d'inconsistance

Richard Ostrowski[1] **Lionel Paris**[2]

[1] Université de Provence (Aix-Marseille 1)
[2] Université Paul Cézanne (Aix-Marseille 3)
LSIS UMR 6168
{richard.ostrowski,lionel.paris}@lsis.org

Résumé

En 1997, B. Selmam et H. Kautz ont proposé une série de 10 challenges. L'un d'entre eux était l'élaboration d'une méthode de recherche locale pour la preuve d'inconsistance d'une formule (challenge 5). Dix ans plus tard, seules quelques pistes ont été explorées. Dans cet article, nous proposons un algorithme en deux phases pour prouver l'insatisfaisabilité de formules sous forme CNF. La première étape consiste à détecter différentes fonctions booléennes (équivalences, portes «et») de manière incrémentale. Ensuite nous utilisons les propriétés de ces différentes portes afin de prouver l'inconsistance des formules (un algorithme polynomial pour les équivalences, des règles de production pour les portes «et»). Nous montrons que cette technique offre de bons résultats par rapport aux approches existantes.

Abstract

In 1997, B. Selman and H. Kautz proposed a series of 10 challenges. One of them concerned the design of a practical stochastic local search procedure for proving unsatisfiability (Challenge 5). Today, more than 10 years later, only few attempts were led to address this challenge, in spite of the great number of incomplete methods for proving satisfiability. In this paper, we propose a two steps algorithm for proving unsatisfiability of CNF formulas. The first step consists in detecting $\Leftrightarrow$-gates greedily. At the same time, a polynomial algorithm is used to eventually prove the unsatisfiability of the extracted set of $\Leftrightarrow$-gates. We show that this method outperforms the existing ones on some classes of instances. This method is then extended to others logical functions ($\wedge$-gates & $\vee$-gates).

1 Introduction

Ces dernières décennies, de nombreuses méthodes incomplètes variées ont été conçues dans le domaine de la programmation par contraintes. Certains d'entre eux traitent des problèmes de décision (problèmes NP-complets), d'autres de problèmes d'optimisation (problèmes NP-difficiles). On peut citer GSAT, Novelty, Novelty+, R-Novelty, R-Novelty+, Adapt-Novelty... (voir [10] pour un état de l'art détaillé). Elles peuvent aussi être utilisées pour extraire des structures particulières des instances, comme les ensembles Strong Backdoor [13] ou les MUS [8] (tout deux des problèmes NP-difficiles). Toutes ces méthodes ont l'avantage d'être efficace en pratique et sont, dans certains cas, le seul moyen de traiter les problèmes les plus gros et les plus difficiles.

Cependant, une infime minorité d'entre elles sont conçues pour prouver l'insatisfaisabilité (problème Co-NP), en dépit du challenge essentiel que cela représente. Un challenge allant dans ce sens fut proposé en 1997 par Selman et Kauz [17] : challenge 5 : "Design a practical stochastic local search procedure for proving unsatisfiability". Dans ce challenge il est question de concevoir une méthode de recherche locale stochastique pour prouver l'inconsistance. Il fut soumis à la communauté et était supposé durer 5 - 10 ans. Mais 12 ans plus tard, aucune réponse satisfaisante à ce challenge n'a encore été proposée, qu'il s'agisse de méthode de recherche locale ou de méthode incomplète en général.

Dans la communauté CSP, quelques tentatives ont été proposées. Elles sont pour la plupart basées sur des propriétés de coloration de la micro-structure des problèmes [4], sur l'utilisation de filtrage par consistances partielles

diverses (consistance d'arc, de chemin, consistance d'arc singleton, k-consistance...) ou basées sur l'exploitation de symétries et de propriété de dominance [3]. D'un autre côté, dans la communauté SAT, les seuls tentatives sont basées sur l'utilisation restreinte de la règle des résolvantes ou des règles de productions permettant de déduire la clause vide (GUNSAT [1], RANGER [15]). Malheureusement, toutes ces méthodes souffrent du passage à l'échelle. Même si certaine d'entre elle se comporte relativement bien sur de petites instances, leurs performances se détériorent de manière dramatique lorsque la taille des instances croît.

Dans ce travail, nous présentons une nouvelle méthode incomplète pour prouver l'inconsistance d'une formule SAT sous forme normal conjonctive (CNF). Alors que la plupart des méthodes déjà proposées utilisent largement la règle de résolution de Robinson [16] ou la règle d'hyper résolution binaire [2], notre méthode comporte deux étapes. La première consiste à détecter puis extraire des fonctions booléennes dans la formule CNF des instances SAT. Ces fonctions (ou portes) sont de la forme $y = f(x_1, x_2, \ldots, x_{n-1})$ (où f = $\Leftrightarrow$ dans un premier temps). Ce processus d'extraction est très coûteux en temps ($O(2^{n-1}$ pour une fonction $y = f(x_1, x_2, \ldots, x_{n-1}))$, c'est pourquoi seulement une sélection de portes sont recherchées. Dans la seconde étape, lorsque des fonctions ont été identifiées, un processus de méta-raisonnement sur l'ensemble de ces fonctions est utilisé pour détecter une éventuelle contradiction. Ce processus est réalisé en temps polynômial par rapport à la taille de l'ensemble des fonctions et est complet sur ce dernier. Cette nouvelle méthode diffère complètement des approches précédemment proposées dans le sens où tout le raisonnement fait sur les portes-$\Leftrightarrow$ est fait en espace polynômial. Il n'y a aucun risque d'explosion combinatoire du nombre de portes, contrairement à l'utilisation de la règle des résolvantes.

Nous avons ensuite étendu ce mode d'extraction et d'exploitation à deux autres fonctions booléennes, les portes-$\wedge$ et les portes-$\vee$. Dans ce cas, le processus de raisonnement sur un ensemble de ces fonctions n'est pas complet. Mais nous avons exhibé quelques propriétés qui de temps en temps (relativement souvent) permettent de détecter l'insatisfaisabilité ou de déduire des littéraux à propager avec une complexité en temps et en espace linéaire.

Nous montrons que les méthodes que nous proposons se comportent très bien par rapport aux méthodes existantes pour prouver l'insatisfaisabilité.

Le papier est organisé comme suit : tout d'abord, nous rappelons les définitions basiques et les notations du problème SAT et des méthodes de déduction. Ensuite nous présentons les deux étapes de notre méthode avec les portes-$\Leftrightarrow$. D'un côté, nous présentons la politique de choix des portes à chercher que nous avons utilisé, ainsi que leur extraction. D'un autre côté, nous exposons l'étape de raisonnement sur l'ensemble des portes extraites. Nous montrons dans la section suivante l'extension de cette méthode aux portes-$\wedge$ et portes-$\vee$. Dans la section d'après, nous décrivons les résultats expérimentaux que nous avons obtenus en utilisant notre approche face à la meilleure des approches existante (GUNSAT). Finalement, une conclusion et des extensions possibles de ce travail sont présentés.

2 Définitions préliminaires

Dans cette section nous rappelons les définitions basiques du problème SAT et les notations que nous utiliserons. Nous donnons aussi des notions de déduction logique (règles d'inférence) qui sont généralement utilisées pour tester l'insatisfaisabilité. Ceci nous permettra de mettre en évidence les différences entre notre méthode et les méthodes existantes.

2.1 Le problème du test de satisfaisabilité (SAT)

Soit $\mathcal{B}$ un langage propositionnel (*i.e.* booléen) composé de formules construites de façon standard, en utilisant les connecteurs usuels ($\vee$, $\wedge$, $\neg$, $\Rightarrow$, $\Leftrightarrow$) et un ensemble de variables propositionnelles. Une *formule CNF* Σ est un ensemble, (interprété comme une conjonction) de *clauses*, où une clause est un ensemble (interprété comme une disjonction) de *littéraux*. Un littéral est une occurrence positive ou négative d'une variable propositionnelle. On représente par $\sim l$ le littéral complémentaire d'un littéral l (si l est un littéral positif d'une variable v, $\sim l = \neg v$, sinon $\sim l = v$).

Rappelons que toute formule booléenne peut être réécrite sous la forme d'une CNF en utilisant le codage de Tseitin [18]. La taille d'une formule CNF Σ est définie par $\sum_{c \in \Sigma} |c|$ où $|c|$ est le nombre de littéraux de la clause c. Une clause *unitaire* (resp. *binaire*) est une clause de taille 1 (resp. 2). Un *monolittéral* est l'unique littéral d'une clause unitaire. On note $nbVar(\Sigma)$ (resp. $nbCla(\Sigma)$) le nombre de variables (resp. clauses) de Σ. $\mathcal{V}(\Sigma)$ (resp. $\mathcal{L}(\Sigma)$) est l'ensemble de variables (resp. littéraux) apparaissant dans Σ. L'ensemble $\mathcal{L}(\Sigma)$ est l'union des littéraux positifs $\mathcal{L}^+(\Sigma)$ et des littéraux négatifs $\mathcal{L}^-(\Sigma)$. Un ensemble de littéraux $S \subset \mathcal{L}(\Sigma)$ est consistant si et seulement si $\forall l \in S, \neg l \notin S$.

L'*interprétation* d'une formule booléenne est l'affectation de valeur de vérité $\{vrai, faux\}$ à ses variables. Un littéral l est satisfait (resp. falsifié) par I si l est positif et $I[l] = vrai$ ou l est négatif et $I[l] = faux$ (resp. l est négatif et $I[l] = vrai$ ou l est positif et $I[l] = faux$). Un *modèle* d'une formule est une interprétation qui satisfait la formule. Usuellement, le problème de décision associé au problème SAT consiste à déterminer si une formule

sous forme CNF admet une solution. Ce problème est NP-complet. Dans ce travail, nous étudions le problème complémentaire du problème SAT, qui consiste à déterminer si une formule CNF n'admet aucun modèle. Ce problème est Co-NP.

2.2 Notion de déduction

Lorsque l'on traite l'insatisfaisabilité en logique propositionnelle, l'utilisation de règles d'inférence est presque obligatoire. L'application d'une règle d'inférence sur un ensemble S de clauses produit une nouvelle clause qui ne change pas la satisfaisabilité de l'ensemble original. Si la clause produite est vide ($S \vdash \bot$), alors l'ensemble de clauses est insatisfaisable.

Une des règles d'inférence la plus connue est la règle des résolvantes de Robinson, appelée aussi règle de résolution [16]. Cette règle permet d'inférer une nouvelle clause, appelée résolvante, à partir de deux clauses initiales sous certaines conditions.

Définition 1 (Règle de résolution de Robinson) *Soit $c_1 = (x_1 \vee x_2 \vee \ldots \vee x_n)$ et $c_2 = (\neg x_1 \vee y_2 \vee \ldots \vee y_m)$ 2 clauses. La clause $R = (x_2 \vee \ldots \vee x_n \vee y_2 \vee \ldots \vee y_m)$ est la résolvante obtenue par application de la règle de résolution sur x_1 et $\neg x_1$ (entre c_1 et c_2) : $c_1, c_2 \vdash_{\text{Rob}} R$.*

Si un ensemble de clauses est insatisfaisable, l'application de cette règle un nombre illimité de fois sur cet ensemble de clauses (incluant les clauses produites) jusqu'à saturation est une méthode complète pour prouver l'insatisfaisabilité. Cela signifie que la clause vide sera toujours produite.

L'inconvénient majeur du procédé associé à l'utilisation répétée de la règle de résolution de Robinson est sa complexité spatiale. En fait, la taille des clauses produites ne peut être bornée (sauf par le nombre total de variables de la formule), et leur nombre peut être exponentiel avant de produire la clause vide. C'est la raison pour laquelle plusieurs travaux ont été entrepris dans le but d'affaiblir cette règle pour avoir quelques garanties sur les complexités en temps et en espace. Il en résulte la perte de la complétude. On peut citer la résolution étendue [18], la résolution directionnelle [5] ou l'hyper-résolution binaire [2] par exemple. Les deux solveurs GUNSAT [1] et RANGER [15] sont basés sur différentes combinaisons de ces règles.

Dans notre méthode, nous n'utilisons aucune de ces règles d'inférence. Nous utilisons une seule règle, très simple, basée sur la propagation unitaire pour déduire des clauses. En fait, quand nous voulons tester la présence d'une clause, nous exploitons la propriété suivante :

Propriété 1 *Soit S un ensemble de clauses et $c = l_1 \vee l_2 \vee \ldots \vee l_n$ une clause. $S \vdash c$ si et seulement si $S \wedge \neg c \vdash \bot$. i.e. $S \vdash l_1 \vee l_2 \vee \ldots \vee l_n$ si et seulement si $S \wedge \neg l_1 \wedge \neg l_2 \wedge \ldots \wedge \neg l_n \vdash \bot$.*

Mais nous n'utilisons que la propagation unitaire (PU) pour tester la satisfaisabilité de l'ensemble $S \wedge \neg l_1 \wedge \neg l_2 \wedge \ldots \wedge \neg l_n$. ($S \wedge \neg l_1 \wedge \neg l_2 \wedge \ldots \wedge \neg l_n \vdash_{\text{PU}} \bot$) qui peut être réalisée en temps linéaire et espace constant.

2.3 Fonctions booléennes

Dans ce papier, nous nous intéressons principalement à 3 fonctions booléennes : portes-$\Leftrightarrow$, portes-$\wedge$ et portes-$\vee$.

Définition 2 (Fonctions booléennes (ou portes)) *On appelle fonction booléenne (ou porte) une formule propositionnelle notée $y = f(x_1, x_2, \ldots, x_{n-1})$, où f est un des connecteurs appartenant à $\{\Leftrightarrow, \oplus, \wedge, \vee\}$ et y et x_i sont des littéraux.*

Définition 3 (Littéraux d'entrées et de sortie) *Dans une porte écrite sous la forme $y = f(x_1, x_2, \ldots, x_{n-1})$, on dit que y est le littéral de sortie de la porte et que tous les x_i sont les littéraux d'entrées de la porte.*

3 Prouver l'insatisfaisabilité avec les portes-$\Leftrightarrow$

Nous décrivons dans cette section un algorithme en deux phases. Étant donné une formule Σ, la première étape consiste à identifier les portes-$\Leftrightarrow$. La seconde consiste à raisonner sur l'ensemble des portes extraites lors de la première phase. Mais commençons par la définition précise des portes-$\Leftrightarrow$.

3.1 Définition

Une porte-$\Leftrightarrow$ $l_1 =\Leftrightarrow (l_2, l_3, \ldots, l_n)$ peut être réécrite comme $(l_1 \Leftrightarrow l_2 \Leftrightarrow \ldots \Leftrightarrow l_n)$. Cette porte, appelée aussi chaîne d'équivalences, furent introduites et étudiées dans [6]. Une porte-$\Leftrightarrow$ de longueur n peut être codée de manière unique par un ensemble de clauses contenant 2^{n-1} clauses de longueur n.

Exemple 1 *Représentation clausale d'une porte-$\Leftrightarrow$ de longueur 3 :*
$l_1 =\Leftrightarrow (l_2, l_3)$ ($l_1 \Leftrightarrow l_2 \Leftrightarrow l_3$) est codée par :
- $l_1 \vee l_2 \vee l_3$
- $l_1 \vee \neg l_2 \vee \neg l_3$
- $\neg l_1 \vee l_2 \vee \neg l_3$
- $\neg l_1 \vee \neg l_2 \vee l_3$

Ces fonctions booléennes possèdent des propriétés très intéressantes :

Propriété 2

$$l_1 =\Leftrightarrow (l_2, l_3) \text{ est équivalent à } l_2 =\Leftrightarrow (l_3, l_1) \quad (1)$$

$$\neg l_1 =\Leftrightarrow (\neg l_2, \neg l_3) \text{ est équivalent à } \neg l_1 =\Leftrightarrow (l_2, l_3) \quad (2)$$

$$\neg l_1 =\Leftrightarrow (l_2, l_3) \text{ est équivalent à } l_1 =\Leftrightarrow (l_2, \neg l_3) \quad (3)$$

$$l_1 =\Leftrightarrow (l_2), l_3 =\Leftrightarrow (l_1, l_4) \text{ est remplacé par } l_3 =\Leftrightarrow (l_2, l_4) \quad (4)$$

$$l_1 =\Leftrightarrow (l_2, l_2, l_3) \text{ est équivalent à } l_1 =\Leftrightarrow (l_3) \quad (5)$$

$$l_1 =\Leftrightarrow (\neg l_1) \vdash \bot \quad (6)$$

Description :

1. *L'opérateur $\Leftrightarrow$ est commutatif et associatif.*
2. *Les couples de négations peuvent être supprimés, ainsi, toute chaîne est équivalente à une chaîne contenant au plus une négation.*
3. *La négation éventuelle peut être placée sur n'importe quel littéral.*
4. *On peut remplacer un littéral d'entrée d'une fonction par tous les littéraux d'entrées d'une autre fonction dont ce littéral est littéral de sortie.*
5. *Les paires d'occurrences d'un même littéral peuvent être supprimées.*
6. *Une chaîne d'équivalences binaire contenant un littéral et son opposé est contradictoire.*

Si une instance ne contient que des portes-$\Leftrightarrow$, elle peut être résolue en temps polynomiale. Certains solveurs savent exploiter ces portes-$\Leftrightarrow$ pour effectuer des simplifications comme pré-traitements.(lsat [12],march_dl [9],eqsatz [11]).

Remarque 1 *Les portes-$\Leftrightarrow$ sont similaires aux fonctions XOR (portes-$\oplus$) du fait de ces propriétés :*

- $l_1 \Leftrightarrow l_2 \Leftrightarrow ... \Leftrightarrow l_n = \neg l_1 \oplus l_2 \oplus ... \oplus l_n$ *ssi* $n = 2*k$, $k \in \mathbb{N}$.
  En d'autres termes, $l_1 =\Leftrightarrow (l_2, l_3, \ldots, l_n)$ *est équivalent à* $\neg l_1 = \oplus(l_2, l_3, \ldots, l_n)$ *ssi* $n = 2*k, k \in \mathbb{N}$
- $l_1 \Leftrightarrow l_2 \Leftrightarrow ... \Leftrightarrow l_n = l_1 \oplus l_2 \oplus ... \oplus l_n$
  ssi $n = 2*k+1, k \in \mathbb{N}$.
  En d'autres termes, $l_1 =\Leftrightarrow (l_2, l_3, \ldots, l_n)$ *est équivalent à* $l_1 = \oplus(l_2, l_3, \ldots, l_n)$ *ssi* $n = 2*k+1, k \in \mathbb{N}$

3.2 Extraction des portes-$\Leftrightarrow$

Il y a plusieurs moyens de détecter les portes-$\Leftrightarrow$. Celui que nous avons retenu commence par choisir un ensemble de variables S de longueur arbitraire suivant une heuristique donnée. Ensuite il énumère toutes les interprétations possibles sur cet ensemble de variables. Pour chaque interprétation, il tente de produire une clause par propagation unitaire en utilisant la propriété 1. *i.e.* $\Sigma, \neg S \vdash_{\text{PU}} \bot$?

À chaque étape, un compteur pour chaque porte possible est maintenu (il n'y en a que 2, la chaîne positive($y =\Leftrightarrow (x_1..x_n)$) et celle négative ($\neg y =\Leftrightarrow (x_1..x_n)$)). Selon la clause qui est produite, l'un ou l'autre de ces compteurs est incrémenté (en fonction du nombre de littéraux négatifs) dans le but de compter le nombre de clauses appartenant aux définitions possibles qui peuvent être produites. Si un de ces compteurs est égal à la moitié du nombre d'interprétations possibles, alors la porte-$\Leftrightarrow$ est détectée. Sinon, si une clause ne peut être dérivée par propagation unitaire, toutes les autres clauses appartenant à la même définition de la porte ne seront pas testées.

Cette procédure de détection est répétée jusqu'à ce qu'un nombre de tentatives (nbTentatives) est atteint. Notre algorithme est incrémental. Il commence à chercher des portes de longueur 2. S'il n'en trouve aucune alors il en cherche avec des longueurs de plus en plus grandes jusqu'à atteindre une longueur maximale (longueurMax). Si pour une longueur n donnée, des portes sont identifiées, la seconde phase est effectuée avant de continuer avec une longueur supérieure. La seconde phase consiste à déterminer si l'ensemble des portes détectées est satisfaisable ou pas.

L'algorithme 1 montre le procédé global.

Algorithm 1: Detection-portes-$\Leftrightarrow$(entrée : une CNF Σ, `nbTentatives`,V,`longueurMax`, sortie : $vrai$ si des portes contradictoires ont pu être produites, $faux$ sinon)

```
1  n ← 2
2  portes-∧ ← {∅} ; /* initialize an empty set of gates */
3  tant que (n ≠ longueurMax) faire
4      essai ← 0
5      tant que (essai ≠ nbTentatives) faire
6          S ← sousEnsemble(V,n)
7          ClausesAvecNbLittNegImpairs ← 0 ; /* Nombre de
           clauses contenant un nombre impair de
           littéraux négatifs                          */
8          ClausesAvecNbLittNegPairs ← 0 ; /* Nombre de clauses
           contenant un nombre pair de littéraux
           négatifs                                    */
9          pour tous les (interprétations I possibles sur S) faire
10             si (Σ ∧ I ⊢^BCP ⊥) alors
11                 si (NbLittNeg(I)% 2) = 0 ) alors
12                     ClausesAvecNbLittNegPairs ←
                       ClausesAvecNbLittNegPairs +1
13                 sinon
14                     ClausesAvecNbLittNegImpairs ←
                       ClausesAvecNbLittNegImpairs +1
15             si (ClausesAvecNbLittNegPairs=n × 2^(n-1)) alors
16                 ajouter(portes-⇔,O,n)
17             si (ClausesAvecNbLittNegImpairs=n × 2^(n-1)) alors
18                 ajouter(portes-⇔,1,n)
19         essai ← essai+1
20     si testSatisfaisabilité(portes-⇔) = vrai alors retourner vrai sinon
       n ← n + 1
21 retourner Faux ;
```

la fonction *testSatisfaisabilité(portes-$\Leftrightarrow$)* est décrite dans la section suivante.

3.3 Raisonner avec les portes-⇔

Comme il a été dit précédemment, il existe un algorithme de complexité polynomiale pour tester la satisfaisabilité d'un ensemble de portes-⇔ [6]. Le processus, qui correspond à la fonction *testSatisfaisabilité(portes-⇔)* de l'algorithme 1 est très simple. Tout d'abords, il faut déplacer l'éventuelle négation de chaque chaîne sur le littéral de sortie, de telle sorte qu'il ne reste plus que des portes-⇔ avec des littéraux d'entrée positifs (en utilisant les propriétés 2.2 et 2.3. Ensuite, en utilisant la propriété 2.4, on choisit une porte ayant un littéral positif en entrée puis on injecte sa définition (les littéraux d'entrées) dans toutes les autres portes contenant ce littéral en entrée. On supprime la porte choisie. Puis on simplifie toutes les nouvelles portes obtenues en utilisant les propriétés 2.5 et 2.6. Si $\perp$ est produite, alors l'ensemble de portes-⇔ est insatisfaisable (et donc toute la formule de départ), sinon on recommence avec une autre porte et un autre littéral jusqu'à ce que chaque littéral ai été traité. Si $\perp$ n'est jamais produite, alors l'ensemble est satisfaisable.

4 Prouver l'insatisfaisabilité avec les portes-∧

Nous allons voir dans cette partie comment implémenter le même type d'algorithme afin de détecter des portes $\wedge$ et $\vee$ (portes-∧ et portes-∨).

4.1 Définitions

Tout comme les portes-⇔, les portes-∧ ont une représentation clausale minimale. Pour une porte de taille n, le nombre minimum de clauses pour représenter une telle fonction est composée d'une clause de taille n contenant le littéral de sortie et les littéraux complémentaires des littéraux d'entrées et $n-1$ clauses binaires contenant le littéral complémentaire du littéral de sortie et d'un des $n-1$ littéraux d'entrés.

Exemple 2 *Représentation clausale minimale d'une porte-∧ de taille 3 :*
$l_1 = \wedge(l_2, l_3)$ *représenté par :*
- $l_1 \vee \sim l_2 \vee \sim l_3$
- $\sim l_1 \vee l_2$
- $\sim l_1 \vee l_3$

Il y a une dualité entre les portes-∧ et les portes-∨ :

Proposition 1

$$l_s = \wedge(l_{l1}, ..., l_n) = \neg\neg(l_s = \wedge(l_1, ..., l_n)) = \neg(\neg l_s \neq \vee(\neg l_1, ..., \neg l_n))$$

Dans ce cas, si nous traitons les portes-∧, il n'est pas nécessaire de traiter les portes-∨ car nous ne pourrons pas inférer plus de choses.

4.2 Extraction des portes-∧

La première étape, tout comme pour les portes-⇔, consiste à identifier les clauses appartenant à la définition de la porte considérée. Malheureusement, l'utilisation de la propagation unitaire, nous permettra d'identifier ces clauses que si elles apparaissent sous la forme minimale (une clause de taille n et $n-1$ clauses binaires). Ce que nous voulons ici, c'est de pouvoir retrouver des portes-∧ qui ne sont pas explicitement présentes dans la formule comme celles détectées dans [7]. Nous devons, pour cela, connaitre le nombre exact de clauses appartenant à la définition. Pour une porte-∧ comportant $n-1$ littéraux d'entrées, 2^{n-1} clauses sont nécessaires. (Ceci s'explique par la dualité entre les portes-∧ et portes-∨). Toutes ces clauses sont de taille n (le littéral de sortie et les $n-1$ littéraux d'entrés). Il est possible de retrouver la forme minimale en utilisant la règle de Résolution 1 sur les $2^{n-1}-1$ clauses.

Exemple 3 *Forme clausale étendue d'une porte-∧ de taille 3 :*
$l_1 = \wedge(l_2, l_3)$ *est représenté par :*
- $l_1 \vee \sim l_2 \vee \sim l_3$
- $\sim l_1 \vee l_2 \vee \sim l_3$
- $\sim l_1 \vee l_2 \vee l_3$
- $\sim l_1 \vee \sim l_2 \vee l_3$

Etant donné un ensemble de variables, combien de définitions différentes de portes-∧ sont possibles ?

Proposition 2 *Pour n variables, il y a exactement $n.2^n$ définitions différentes de portes-∧.*

Preuve 1 *Pour chaque variable $V_i, i \in \{1, \ldots, n\}$ nous associons son littéral positif (l_i) et son littéral négatif ($\neg l_i$). Si nous considérons le littéral de sortie (positif), il est possible d'avoir toutes les combinaisons différentes de définitions construites sur les $n-1$ littéraux d'entrés. Nous en avons exactement le même nombre si l'on considère le littéral de sortie (négatif). Donc pour une variable de sortie $V_i, i \in \{1, \ldots, n\}$, nous avons exactement 2^n définitions. Enfin, si nous considérons toutes les variables de sortie (n), nous obtenons $n * 2^n$ définitions possibles.*

Proposition 3 *Etant donné une clause $c = l_1 \vee \cdots \vee l_n$, le nombre de portes-∧ contenant la clause c dans leur définition est égal à $n * 2^{n-1}$.*

Preuve 2 *Soit $nb(i)$ le nombre de portes-∧ contenant la clause numéro i dans leur définition. On remarque que la somme de tous ces nombres pour chaque clause ($\sum_{i=1}^{2^n} nb(i)$) est égal au nombre possible de portes-∧ sur n littéraux ($n.2^{n-1}$), multiplié par le nombre de clauses de leur définition (2^n). Nous avons donc $\sum_{i=1}^{2^n} nb(i) = n.2^{n-1}.2^n$. Maintenant, nous remarquons que chaque clause apparait dans le même nombre de définitions de*

portes-$\wedge$, donc $nb(i)$ est le même pour chaque clause et donc $\sum_{i=1}^{2^n} nb(i) = 2^n.nb(i)$. On en conclut que, $nb(i) = n.2^{n-1}$.

Le processus d'extraction des portes-$\wedge$ est similaire à celui des portes-$\Leftrightarrow$, mais en considérant la représentation sous forme clausale étendue des définitions des portes-$\wedge$. Il n'y a que deux différences :

1. L'énumération de toutes les affectations possibles, sur les n variables, ne peut pas être interrompue si une clause ne peut pas être inférée par propagation unitaire (car nous devons tester l'existence des $n.2^n$ portes possibles).
2. Lorsque nous avons testé toutes les possibilités, nous devons extraire toutes les portes-$\wedge$ possibles ($n.2^n$), contrairement aux portes-$\Leftrightarrow$ (deux définitions possibles).

L'algorithme 2 montre le processus d'extraction des portes-$\wedge$. Dans cet algorithme, le tableau `nbClauses` va comptabiliser le nombre de clauses identifiées pour chaque définition possible de portes-$\wedge$ sur l'ensemble $\mathcal{S}$ de variables. A chaque fois qu'une clause peut être inférée par PU dans Σ ($\Sigma \wedge \mathcal{I} \vdash^{\text{UP}} \bot$), la fonction `mettreAJour` incrémente de un les définitions de portes-$\wedge$ concernées.

Algorithm 2: Detection-portes-$\wedge$(entrée : Σ, `nbtentatives`,V,`longueurMax`, sortie : $vrai$ si des portes contradictoires ont pu être produites, $faux$ sinon)

```
1  n ← 2
2  portes-∧ ← {∅} ; /* initialisation de l'ensemble des
   portes à vide                                          */
3  tant que (n ≠ longueurMax) faire
4      essai ← 0
5      tant que (essai ≠ nbtentatives) faire
6          S ← sous-ensemble(V,n)
7          nbClauses[n × 2^n] ← [0,...,0]; /* nbClauses est un
           tableau de taille (n × 2^n) rempli de 0        */
8          pour tous les (interprétations I possibles sur S) faire
9              si (Σ ∧ I ⊢^UP ⊥) alors mettreAJour(I,nbClauses)
10         pour tous les (i dans 1,...,n × 2^n) faire
11             if (nbClauses[i]=n × 2^(n-1)) then add(∧-gate,i)
12         essai ← essai+1
13     ensembleMonoLit ← SimplifierPortes(portes-∧)
14     si (ensembleMonoLit ≠ ∅) alors
15         Σ ← UP(Σ ∪ ensembleMonoLit)
16         si (Σ = ∅) alors
17             retourner faux ; /* Σ est satisfaisable    */
18         sinon si (□ ∈ Σ) alors
19             retourner vrai ; /* Σ est insatisfaisable  */
20         sinon
21             n ← 2
22     sinon n ← n + 1
23 retourner faux
```

4.3 Raisonner avec les portes-$\wedge$

La seconde étape consiste à simplifier ces portes afin de trouver une contradiction. Malheureusement, il n'existe pas d'algorithme polynomial afin de tester la satisfaisabilité d'un ensemble de portes-$\wedge$. Cependant, certaines propriétés peuvent être utilisées afin de simplifier cet ensemble.

Propriété 3

$$l_s = \wedge(l_1, \ldots, l_k), l_s = \wedge(l_{k+1}, \ldots, l_n) \vdash l_s = \wedge(l_1, \ldots, l_k, l_{k+1}, \ldots, l_n) \quad (7)$$

$$l_s = \wedge(l_1, \ldots, l_k, \neg l_k) \vdash \neg l_s \quad (8)$$

$$l_s = \wedge(\neg l_s, l_1, \ldots, l_k) \vdash \neg l_s \quad (9)$$

$$l_s = \wedge(l_1, \ldots, l_k), l_s \vdash l_1, \ldots, l_k \quad (10)$$

$$l_s = \wedge(l_1, \ldots, l_k, l_{k+1}, \ldots, l_n), \neg l_s = \wedge(l_1, \ldots, l_k, l'_{k+1}, \ldots, l'_n) \vdash l_1, \ldots, l_k \quad (11)$$

$$l_s = \wedge(l_1, \ldots, l_k), \neg l_s = \wedge(l_1, \ldots, l_k) \vdash \bot \quad (12)$$

$$l_s = \wedge(l_1, \ldots, l_k), \neg l_s = \wedge(l_1, \ldots, l_k, l_{k+1}) \vdash l_s, l_1, \ldots, l_k, \neg l_{k+1} \quad (13)$$

Preuve 3
- *(7) : propriété de fusion. C'est une règle d'inférence. Tout modèle de la partie gauche est un modèle de la partie droite.*
- *(8) : En considérant la représentation clausale minimale, nous avons les deux clauses $\neg l_s \vee \neg l_k$ et $\neg l_s \vee l_k$. Par résolution, nous obtenons $\neg l_s$.*
- *(9) : En considérant la représentation clausale minimale, nous avons la clause $\neg l_s \vee \neg l_s$.*
- *(10) : Comme l_s est à vrai, nous avons $l_1, \ldots, l_k$ qui sont à vrai.*
- *(11) : Si l_s est à vrai alors $l_1, \ldots, l_k, l_{k+1}, \ldots, l_n$ sont déduits, si l_s est à faux, alors $l_1, \ldots, l_k, l'_{k+1}, \ldots, l'_n$ sont aussi déduits.*
- *(12) : Avec (10), on infère $l_1, \ldots, l_k$ qui falsifie la seconde porte.*
- *(13) : Avec (10), on infère $l_1, \ldots, l_k$ et ensuite l_s. Pour satisfaire la seconde porte, nous devons avoir $\neg l_{k+1}$.*

5 Expérimentations

Les expérimentations ont été réalisées sur des Pentium IV 3.2GHz avec 1 GB de RAM sous linux. Pour chaque problème nous avons comparé notre méthode avec GUNSAT[1] [1].

Nous avons mené deux types d'expérimentations. Nous avons d'abord fait des tests sur des problèmes insatisfaisables générés aléatoirement et ensuite sur des problèmes insatisfaisables structurés issus de la satlib [2].

[1] http ://www.lri.fr/~simon/research/gunsat/gunsat-V1.jar
[2] *http ://www.cs.ubc.ca/~hoos/SATLIB/benchm.html*

Instance	$\Leftrightarrow$			$\wedge$			les deux				GUNSAT	
	%S	# $\Leftrightarrow$	T(s)	%S	#$\wedge$	T(s)	%S	# $\Leftrightarrow$	#$\wedge$	T(s)	%S	T(s)
ssa	25	498	36	25	4415	0.3	50	593	268	0.1	–	–
pret	100	281	0	0	0	0	95	1372	441	2	0	0
jnh	100	70	3.1	100	1913	2.7	89.5	719	273	0.35	57	8.48
dubois	92	217	0	0	0	0	74.6	1051	330	0.69	0	0
aim-50	37.5	3.6	0.07	88.24	11098	9.2	79	741	252	0.35	100	1.41
aim-100	21.4	2	10.81	14.29	172	9.74	80	1089	302	1	55	11.93
aim-200	0	0	0	0	0	0	0	0	0	0	60	63

TAB. 1 – Résultats de notre méthode sur des instances structurées. Chaque ligne du tableau correspond à une classe d'instance dont le nom est donné dans la colonne Instance. La colonne %S donne le % d'instances résolues. La colonne $\Leftrightarrow$ (resp. $\wedge$) donne le nombre moyen de portes-$\Leftrightarrow$ (resp. portes-$\wedge$) détectées pendant la recherche. Enfin, la colonne T(s) donne le temps moyen de résolution en secondes. Chaque problème a été testé 10 fois.

Nous avons évalué trois versions différentes de notre méthode. La première, correspondant à la colonne $\Leftrightarrow$ dans le tableau 1 et 2, est l'implantation de la méthode décrite dans 3 (portes-$\Leftrightarrow$). La seconde, correspondant à la colonne $\wedge$ des tableaux 1 et 2, est l'implantation de la méthode décrite dans 4 (portes-$\wedge$). La dernière, correspondant à la colonne *les deux* des deux tableaux, est l'implantation des deux méthodes.

L'heuristique utilisée pour choisir les n variables, pour chacune des méthodes, afin de détecter des portes, fonctionne comme suit : si l'on recherche une porte de longueur n, et qu'il existe des clauses de taille n alors l'ensemble des variables de cette clause est considéré. S'il n'en existe pas ou si toutes les clauses de la taille considérée ont été testées, alors l'heuristique génère aléatoirement un ensemble de n variables parmi $10.n$ variables ayant le plus grand nombre d'occurrences dans la formule. Ceci est effectué 1000 fois.

Nous remarquons que sur les instances aléatoires ainsi que les structurés, notre méthode est meilleure que GUNSAT et que la combinaison de détection des portes-$\Leftrightarrow$ et portes-$\wedge$ donne de bons résultats.

6 Conclusion et perspectives

Dans cet article, nous avons vu comment utiliser la propagation unitaire afin de déduire des clauses d'une formule Σ. La détection de ces clauses permet d'identifier des fonctions booléennes. Après avoir identifié un certain nombre de portes (portes-$\Leftrightarrow$ et portes-$\wedge$), nous avons vu comment les utiliser afin de développer une méthode incomplète en deux phases pour la preuve de l'inconsistance. Les résultats obtenus sont très encourageants. Notre méthode se montre plus performante que GUNSAT sur de nombreuses classes d'instances. Ceci montre encore l'importance d'exploiter la structure cachée des différents problèmes. Ces bons résultats mettent en valeur les travaux de Pham *et al.* [14]. Dans ces travaux, les auteurs exploitent la structure des problèmes dans une méthode de recherche locale. Cette méthode était la première capable de résoudre les problèmes parity-32.

Ces travaux ouvrent de nombreuses perspectives. D'abord, il serait intéressant de voir s'il est possible de trouver d'autres règles d'inférences concernant les différentes portes afin de simplifier la partie fonctionnelle. Les propriétés proposées, concernant les portes-$\wedge$ permettent de produire de nombreux mono-littéraux. Il serait intéressant de voir si une telle détection et simplification peut être utilisée comme pré-traitement. Enfin, sur les portes-$\Leftrightarrow$, il existe aussi des propriétés qui ne sont pas utilisés afin de produire des mono-littéraux.

Références

[1] Gilles Audemard and Laurent Simon. Gunsat : a greedy local search algorithm for unsatisfiability. In *International Joint Conference on Artificial Intelligence(IJCAI'07)*, pages 2256–2261, jan 2007.

[2] Fahiem Bacchus. Enhancing davis putnam with extended binary clause reasoning. In *Eighteenth national conference on Artificial intelligence*, pages 613–619, Menlo Park, CA, USA, 2002. American Association for Artificial Intelligence.

[3] Belaïd Benhamou and Mohamed Réda Saïdi. A new incomplete method for csp inconsistency checking. In *Twenty-Third AAAI Conference on Artificial Intelligence (AAAI-08)*, Chicago, Illinois, USA, juillet 2008. AAAI Press. à paraître.

[4] J.N. Bès and P. Jégou. Proving graph un-colorability with a consistency check of csp. In *Proceedings of the 17th IEEE International Conference on Tools with Artificial Intelligence*, pages 693–694, Hong Kong, China, novembre 2005. IEEE. Poster.

Aléatoire		⇔		∧		les deux		GUNSAT	
$nbVar$	$nbCla/nbVar$	%S	T(s)	%S	T(s)	%S	T(s)	%S	T(s)
50	4.25	100	0.25	99	0.26	99	0.26	58	60
50	5	100	0.12	100	0.11	100	0.11	86	18
50	6	100	0.05	100	0.04	100	0.04	97	5
60	4.25	100	0.95	100	1.46	100	1.52	35	126
60	5	100	0.23	100	0.28	99	0.28	68	67
60	6	100	0.1	100	0.1	100	0.1	92	16
70	4.25	100	4.77	99	16	99	15.4	23	189
70	5	100	0.54	99	0.9	99	0.88	51	187
70	6	100	0.16	100	0.22	100	0.21	87	59
80	4.25	100	13.6	52.5	58.4	88.5	70	–	–
80	5	100	1.49	100	3.56	100	3.48	–	–
80	6	100	0.32	100	0.57	99	0.56	79	146

TAB. 2 – Résultats de notre méthode sur les instances aléatoires (clauses de longueur 3). Chaque ligne représente une catégorie d'instance caractérisée par son nombre de variables et de clauses. Chaque catégorie contient 20 problèmes et chaque problème est testé 10 fois. La colonne %S est le pourcentage d'instances résolues et la colonne T(s) donne le temps moyen de résolution en seconde.

[5] Rina Dechter and Irina Rish. Directional resolution : The davis-putnam procedure, revisited. In *In Proceedings of KR-94*, pages 134–145. Morgan Kaufmann, 1994.

[6] B. Dunham and H. Wang. Towards feasible solutions of the tautology problem. *Annals of Math. Log.*, 10 :117–154, 1976.

[7] É. Grégoire, B. Mazure, R. Ostrowski, and L. Saïs. Automatic extraction of functional dependencies. In *proc. of SAT*, volume 3542 of *LNCS*, pages 122–132, 2005.

[8] Eric Grégoire, Bertrand Mazure, and Cédric Piette. Local-search extraction of muses. *Constraints Journal*, 12(3) :325–344, 2007.

[9] Marijn Heule, Joris van Zwieten, Mark Dufour, and Hans van Maaren. March_eq : Implementing additional reasoning into an efficient lookahead sat solver. In *Seventh International Conference on Theory and Applications of Satisfiability Testing (SAT2004)*, pages 345–359, 2004.

[10] Holger H. Hoos and Thomas Stützle. *Stochastic Local Search : Foundations and applications*. Elsevier / Morgan Kaufmann, San Francisco, CA, 2004.

[11] Chu Min Li. Integrating equivalency reasoning into Davis-Putnam procedure. In *proceedings of Conference on Artificial Intelligence AAAI*, pages 291–296, Austin, USA, 2000. AAAI Press.

[12] R. Ostrowski, E. Grégoire, B. Mazure, and L. Saïs. Recovering and exploiting structural knowledge from CNF formulas. In *CP'02*, pages 185–199, 2002.

[13] Lionel Paris, Richard Ostrowski, Pierre Siegel, and Lakhdar Saïs. Computing and exploiting Horn strong backdoor sets thanks to local search. In *Proceedings of the 18th International Conference on Tools with Artificial Intelligence (ICTAI'2006)*, pages 139–143, Washington DC, United States, 2006.

[14] Duc Nghia Pham, John Thornton, and Abdul Sattar. Building Structure into Local Search for SAT. In *Proceedings of IJCAI'07*, pages 2359–2364, Hyderabad, India, January 2007.

[15] Steven Prestwich and Inês Lynce. Local search for unsatisfiability. In *In Proceedings of SAT*, pages 283–296. Springer, 2006.

[16] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM (JACM)*, 12(1) :23–41, January 1965.

[17] Bart Selman, Henry A. Kautz, and David A. McAllester. Ten challenges in propositional reasoning and search. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence (IJCAI'97)*, pages 50–54, 1997.

[18] G.S. Tseitin. On the complexity of derivations in the propositional calculus. In H.A.O. Slesenko, editor, *Structures in Constructives Mathematics and Mathematical Logic, Part II*, pages 115–125, 1968.

Pourquoi les solveurs SAT modernes se piquent-ils contre des cactus ?*

Gilles Audemard[1] **Mouny Samy Modeliar**[1] **Laurent Simon**[2]

[1]Univ. Lille-Nord de France
CRIL/CNRS UMR8188
Lens, F-62307 CRIL-CNRS,

{audemard,modeliar}@cril.fr

[3] Univ. Paris-Sud
LRI/CNRS UMR 8623
INRIA Saclay
Orsay, F-91405

simon@lri.fr

Abstract

Malgré les progrès impressionnants observés ces dernières années autour de la résolution pratique d'instances SAT industrielles, les solveurs CDCL (*Conflict Driven, Clause Learning*) ont toujours un comportement global qui se doit d'être relativisé. Celui-ci est en effet souvent caractérisé par les figures *cactus*, qui suggèrent l'existence d'un seuil d'incompétence atteint par tout solveur après un certain temps de recherche. Les chances de résoudre une instance une fois passée cette limite semble décroître de manière exponentielle.

Dans cet article, nous rapportons une étude expérimentale intensive sur un solveur CDCL canonique, en nous focalisant sur un panel d'instances industrielles. Nous étudions particulièrement la décroissance du nombre de niveaux de décisions durant la recherche. Cela nous permet de proposer une hypothèse quant à la puissance du mécanisme d'apprentissage, à savoir, la production de clauses particulières, appelées "*clauses glues*". Nous montrons, sur de nombreuses instances, que de moins en moins de clauses glues sont produites, celle ci suivant une loi de type $log(x)^B$, ce qui peut être une bonne explication de l'inefficacité des solveurs passé un certain temps.

1 Introduction

La plupart des articles qui traitent du problème SAT commencent traditionnellement par rappeler les progrès extraordinaires mesurés ces dernières années, notamment sur les problèmes industriels. Les résultats récents sont en effet impressionnants, à tel point qu'aujourd'hui, la résolution de nombreux problèmes industriels passe par un codage SAT à la place des solveurs ad hoc [13] du domaine. Mais le tableau global n'est pas si positif. Si l'on revêt quelques instants le costume de l'avocat du diable, on peut souligner que, dans cette quête de la performance, seuls quelques progrès ont été fait sur la compréhension réelle des mécanismes de ces solveurs. Certes, nous savons quels sont les ingrédients nécessaires pour obtenir de bonnes performances, mais leurs rôles respectifs ne sont pas encore parfaitement connus. Si l'on suit ce raisonnement, on peut aussi faire remarquer que ces solveurs modernes, descendants de ZCHAFF [10], utilisent des mécanismes d'apprentissages, comme le premier UIP (*Unique Implication Point* [16]), déjà introduits 8 ans avant ZCHAFF [14]. Derrière les progrès certes impressionnants, se cache souvent une simple réécriture de ZCHAFF, accompagnée d'astuces plus ou moins importantes (écriture compacte des clauses, redémarrages rapides, sauvegarde de la phase, littéraux bloqués...).

Les structures de données paresseuses introduites dans ZCHAFF ont provoqué un bouleversement du monde SAT qui dépasse de loin leurs motivations initiales. En effet, ces structures de données, initialement introduites pour empêcher les défauts de cache [17] ont interdit toute information de type *lookahead* (comme le simple comptage d'occurrences des littéraux). Comme ces structures de données sont incontournables pour résoudre les problèmes industriels qui comptent un grand nombre de variables et de clauses, la plupart des solveurs n'ont pas eu d'autres choix que de suivre cette voie, devenant tous de type *lookback*. Dès lors, les variables (pour les choix heuristiques) et les clauses (pour connaître celles qu'il faut supprimer parmi les clauses apprises) sont estimées uniquement sur leur ac-

*supporté par le projet ANR UNLOC

tivité passée. Les solveurs sont donc maintenant conçus pour atteindre le plus rapidement possible les conflits afin d'apprendre de nouveaux nogoods et de mettre à jour les valeurs heuristiques connexes. L'un des effets de bord de ce schéma d'algorithme est que les solveurs sont devenus de plus en plus imprévisibles. Les *anciens* solveurs, basés sur les techniques de type *lookahead* (voir [9] par exemple), étaient beaucoup plus faciles à comprendre. En effet, la recherche était guidée, par exemple, pour réduire le plus possible la taille de l'arbre de recherche. Les techniques de type *lookback*, comme par exemple l'heuristique VSIDS (Variable State Independent Decaying Sum [10]) couplée à des redémarrages rapides [6], sont beaucoup plus difficiles à appréhender. En conclusion, nous savons donc implanter des solveurs très efficaces, qui plus est en moins de 1000 lignes de code, mais nous ne savons pas toujours quels sont les mécanismes qui sont vraiment importants et pourquoi ils le sont. De nombreuses questions restent donc en suspens. Que sont de bonnes clauses apprises ? pourquoi les redémarrages à la luby sont-ils si efficaces ? Est ce que certaines clauses apprises ne servent qu'à mettre à jour les heuristiques et simuler le retour arrière ?

Dans ce contexte, il nous semble de plus en plus important de construire une réelle science expérimentale autour des algorithmes [5] afin de comprendre le comportement des solveurs de type *lookback*. C'est l'idée principale que cet article tente de suivre. Nous proposons une étude expérimentale d'un solveur CDCL canonique (MINISAT [3]) afin d'essayer de comprendre ces forces et faiblesses. Notre travail, est un prolongement de remarques déjà faites dans [1], où nous montrions la relation entre la décroissance des niveaux de décisions et la performance des solveurs.

Notre hypothèse est que les solveurs CDCL réduisent, tout au long de la recherche, le nombre de décisions à faire avant d'atteindre un conflit. De plus, sur de nombreuses instances, cette décroissance des niveaux de décision semble suivre une loi exponentielle (par rapport au nombre de conflits déjà rencontrés), rendant de plus en plus difficile l'obtention du dernier conflit. Nous faisons une autre hypothèse, à savoir que cette loi exponentielle est guidée par l'incapacité des solveurs à garder un taux élevé de production de certaines clauses tout au long de la recherche. Nous montrons cela sur des variantes de MINISAT, suggérant que ces limites sont dues à l'architecture des solveurs CDCL. Néanmoins, il est important de noter que cet article risque de soulever plus de questions que de réponses. Il est en effet utopique de chercher des lois de régression qui puissent s'appliquer sur des problèmes très différents, certains de nos résultats pourront donc être vu comme négatifs. Malgré tout, nous pensons que ce travail est nécessaire pour aider la communauté à mieux comprendre comment les solveurs se comportent.

Le reste de l'article est organisé comme suit. La section 2 présente quelques notations et rappelle les résultats obtenus dans un travail annexe [1]. La troisième section étudie les différentes lois de régression utilisées. Avant de conclure, la section 4 étudie l'évolution des performances des solveurs sur des problèmes difficiles. Les performances sont mesurées par la capacité à produire des clauses que nous pensons importantes.

Algorithm 1: Solveur de type CDCL

```
   Input: Σ une formule CNF
   Output: SAT ou UNSAT
1  I = ∅ ;                    /* interprétation */
2  nd = 0 ;                   /* niveau de décision */
3  x_c = 0 ;                  /* numéro du conflit */
4  while (true) do
5      c = propagationUnitaire(Σ,I);
6      if (c !=null) then
7          x_c = x_c + 1;
8          γ =AnalyseConflit(Σ,I,c);
9          bl = CalculeRetourArriere(γ,I);
10         if (bl < 0) then return UNSAT;
11         Σ = Σ ∪ {γ};
12         if (redémarrage()) then bl = 0;
13         RetourArriere(Σ,I,bl);
14         nd = bl;
15     else
16         if (toutes les variables sont affectées) then
17             return SAT;
18         ℓ = choixLiteralDecision(Σ);
19         nd = nd + 1;
20         I = I ∪ {ℓ};
21
22 end
```

2 Travaux préliminaires et antérieurs

Soit $V = \{x_1, \ldots, x_n\}$ un ensemble de *variables booléennes*, un *littéral* ℓ_i est une variable x_i ou sa *négation* $\neg x_i$. Une *clause* est une disjonction de littéraux $c_i = l_1 \vee l_2 ... \vee l_{n_i}$. Une clause *unitaire* ne contient qu'un seul littéral. Le problème SAT est un problème de décision défini comme suit : étant donné une formule propositionelle Σ sous forme normale conjonctive (CNF, une conjonction de clauses), existe-t-il une affectation des variables V de Σ tel que Σ soit satisfaite, c.a.d., tel que toutes les clauses de Σ soient satisfaites ?

2.1 Solveurs CDCL

L'algorithme 1 montre l'organisation d'un solveur de type CDCL (nous ne rentrerons pas dans le détail de chaque fonction de cet algorithme). Une branche typique d'un tel solveur est une séquence de décisions suivies de propaga-

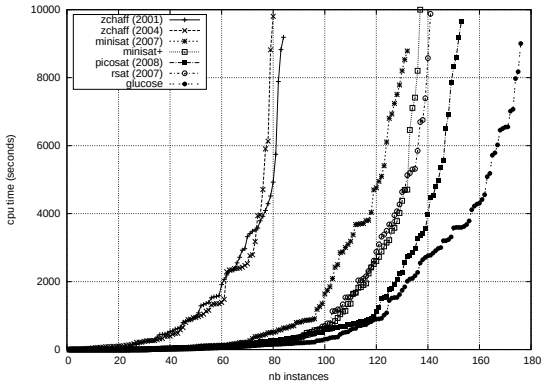


FIG. 1 – Figure cactus sur les 234 instances industrielles de la compétition SAT'07. MINISAT+ est MINISAT avec les redémmarages à la luby, la sauvegarde de phase et les litéraux bloqués. GLUCOSE est notre solveur [1].

Series	#Benchs	% Decr.	$m_{x_{jp}}$	r
een	8	62%	1.1×10^3	0.31
goldb	11	100%	1.4×10^6	0.22
grieu	7	71%	1.3×10^6	0.48
hoons	5	100%	7.2×10^4	0.30
ibm-2002	7	71%	4.6×10^4	0.21
ibm-2004	13	92%	1.9×10^5	0.24
manol-pipe	55	91%	1.9×10^5	0.40
miz	13	0%	–	0.37
schup	5	80%	4.8×10^5	0.38
simon	10	90%	1.1×10^6	0.34
vange	3	66%	4.0×10^5	0.06
velev	54	92%	1.5×10^5	0.25
all	199	83%	3.2×10^5	0.31

TAB. 1 – Décroissance des niveau de décision et justification à postériori

tions, répétée jusqu'à ce qu'un conflit survienne. Chaque littéral de décision (lignes 18–20) est affecté à un niveau de décision (nd), les littéraux qui se déduisent (par propagation unitaire) de cette décision ont le même niveau. Si tous les littéraux sont affectés, alors I est un modèle de Σ (lignes 16-17). À chaque fois qu'un conflit est atteint par propagation unitaire (c est alors la clause falsifiée, lignes 5-6), un nogood γ est calculé (ligne 8) en utilisant une méthode donnée, le plus souvent le premier UIP (Unique Implication Point) [16] et un niveau de backjump bl est calculé. À ce moment, on peut avoir prouvé l'inconsistance de la formule. Si ce n'est pas le cas, on fait un saut arrière et le nouveau niveau de décision devient égal au niveau de backjump (lignes 13-14). Enfin, de temps à autres, les solveurs CDCL forcent des redémarrages (diverses stratégies sont possibles, voir [6] par exemple) et dans ce cas, on remonte tout en haut de l'arbre de recherche (ligne 12).

2.2 Des figures Cactus ?

Les *figures cactus*, comme par exemple la figure 1, sont souvent utilisées comme un outil résumant les performances d'un ensemble de solveurs sur un ensemble de problèmes. Une telle figure montre clairement que tous les solveurs SAT atteignent un seuil d'incompétence I après un certain temps (il ne semble pas illogique de relier ce seuil d'incompétence au principe de PETER [11]). Avec les progrès récents réalisés, ce point d'incompétence est régulièrement poussé vers la droite, mais il existe toujours. Intuitivement, l'existence de ce point semble signifier que, si un solveur n'a pas résolu une instance donnée en I secondes, alors il y a peu de chances qu'il la résolve en $2 \times I$ secondes. La possibilité qu'il la résolve semble ainsi décroître exponentiellement avec le temps additionnel donné, une fois ce seuil d'incompétence franchi.

Dans [4], les auteurs suggèrent que ceci peut être relié aux phénomènes de type longue traîne (*heavy tailed*) observés dans les arbres de recherche. Les auteurs montrent qu'ils peuvent être dûs à de mauvais choix de variables en haut de l'arbre (Cette idée est également reliée à celle des "*backdoors*" d'une formule [15]). Néanmoins, les solveurs modernes utilisent des redémarrages extrêmement rapides (on notera que l'on ne devraient peut-être plus les appeler redémarrage, mais réorganisation des valeurs de littéraux [2]). Si les phénomènes de longues traînes sont effectivement dus à de mauvais choix en haut de l'arbre, ces solveurs devraient donc bien en être préservés.

D'après nos connaissances, il n'y a donc pas de justification théorique satisfaisantes des figures cactus. Nous pensons que, pour améliorer les solveurs CDCL et proposer de nouvelles stratégies, nous devons d'abord comprendre les raisons de leurs inefficacité sur certaines instances. Nous espérons que cet article sera un premier pas dans ce sens.

2.3 Une première observation : les niveaux de décision décroissent

Nous rappelons ici les premières observations faîtes dans [1]. Pour des raisons de simplicité, nous rappelons ici la même table (table 1) légèrement modifiée. Ces observations ont été faites sur de nombreuses instances issues des dernières compétitions SAT. Voici comment nous l'avons obtenu : nous lançons le solveur MINISAT [3] sur ces instances. Chaque fois qu'un conflit numéroté x_c est atteint (ligne 7 de l'algorithme 1), on enregistre le niveau de décision nd où il est survenu. A la fin de la recherche, nous calculons les caractéristiques (a et b) de la loi de régression linéaire $y = a \times x + b$ sur l'ensemble des points (x_c,nd). Dans ce travail préliminaire, nous ne nous sommes intéressés qu'au comportement général de cette droite, et particu-

lièrement à sa décroissance. Nous n'avons pas regardé si cette régression était la meilleure possible, ce qui est l'un des objets de cet article. Si cette droite décroît alors $a < 0$. Nous pouvons dans ce cas "*prédire*" quand le solveur résout l'instance (qu'elle soit satisfaisable ou pas). C'est le moment où la droite de régression intersecte l'axe des x (le conflit est trouvé au niveau de décision 0 de l'arbre de recherche). Nous appelons ce point, le point de "*justification à posteriori*" (connaître tous les points est en effet nécessaire pour calculer la droite de régression et donc la justification). Les coordonnées de ce point sont ($x_{jp} = -b/a$,0). Cette valeur donne une bonne intuition de la décroissance des niveaux de décision durant la recherche, une décroissance rapide donnera des justifications relativement petites.

La table 1 montre que sur une large majorité des instances les niveaux de décision décroissent bel et bien. "#Benchs" donne le nombre de benchmarks dans la série, "#Decr." donne le pourcentage de benchmarks qui voient une décroissance des niveaux de décision. La colonne 4 donne la médiane $m_{x_{jp}}$ des justifications calculées lorsque cela est possible (quand il y a effectivement décroissance). On peut noter que, dans la plupart des cas (167 sur 199), les droites de régression décroîssent. De plus, dans très peu de cas, la valeur $m_{x_{jp}}$ est importante.

La dernière colonne donne l'indice de Bravais-Person r. Cet indice vaut entre 0 et 1. Il montre l'adéquation entre les points (x_c, nd) et la droite de régression. Une valeur au-dessus de 0.71 montre une adéquation pas trop mauvaise, une valeur au dessus de 0.86 une bonne. Une conclusion s'impose : les régressions linéaires calculées sont très mauvaises.

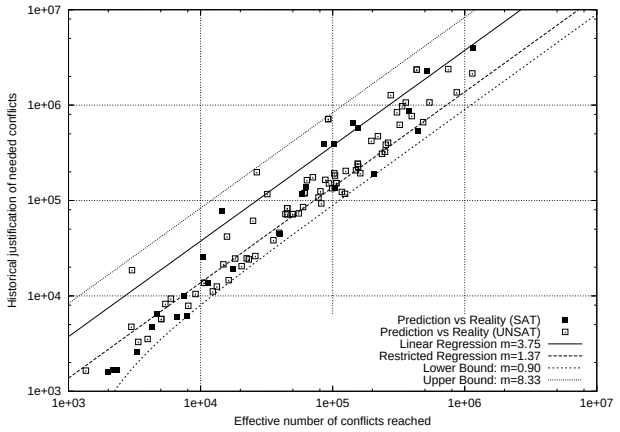


FIG. 2 – Relation entre justification à posteriori et nombre de conflits nécessaires à MINISAT pour résoudre l'instance.

Néanmoins, le fait d'avoir de mauvaises valeurs de r ne veut pas dire que la justification à posteriori est mauvaise. En effet, que pourrait-on déduire si les justifications sont de bonnes estimations du nombre réel de conflits nécessaires pour prouver l'inconsistance (ou plus incroyable encore, une solution) ? Cela voudrait peut-être dire que la décroissance des niveaux est une caractéristique importante et une des puissances des solveurs CDCL : Ils forcent les niveaux de décision à régulièrement décroître le long de la recherche.

La figure 2 (également issue de [1]) est construite comme suit : Chaque point (x, y) correspond à une instance. x correspond au nombre de conflits effectifs nécessaire à MINISAT pour résoudre l'instance donnée, y à la justification à posteriori. Seules les instances résolues en moins de 2 millions de conflits y sont représentées (autrement, le calcul de la justification serait biaisé). Cette figure montre clairement une étroite relation entre les justifications calculées et le nombre réels de conflits nécessaires pour terminer la recherche. La justification à posteriori est comprise entre 0.9 et 8.33 fois le nombre réels de conflits nécessaires à MINISAT pour résoudre le problème. Dans la plupart des cas, la justification est approximativement égale à 1.37 le nombre de conflits nécessaires. Ce qui est assez frappant, c'est que l'on ne peut pas faire de distinctions entre les instances SAT et UNSAT : quand une solution existe, elle n'est pas trouvée soudainement, par chance.

2.4 Notion de "clauses glues"

Dans le même article [1], nous montrons que le nombre de différents niveaux de décision d'une clause apprise, donne une bonne estimation de son utilité. Plus petit est ce nombre, "meilleure" semble être la clause apprise. Il est intéressant de noter que cette mesure est optimale pour la clause assertive obtenue par le premier UIP [7, 8]. Associée à une suppression agressive des "*mauvaises*" clauses apprises, cette mesure a donné lieu à un solveur nommé GLUCOSE [1] qui s'est avéré très efficace dans la pratique.

Il est déjà de notoriété que les clauses unaires et binaires sont importantes, mais les clauses de LBD égal à 2 le sont également (notez d'ailleurs que les clauses binaires ont un LBD égal à 2). Elles contiennent un littéral v du dernier niveau de décision (le littéral assertif) et un ensemble de littéraux d'un autre groupe de littéraux S (celui où sera fait le retour arrière), et après retour arrière, le littéral v sera fusionné (collé) avec le groupe de littéraux S, et ce, quelque soit la taille de la clause, que nous appellerons donc clause *glue*. Avec la technique de sauvegarde de la phase, il est fort possible que les variables de cet ensemble S soient toujours affectées avec les mêmes valeurs et, grâce à cette clause *glue*, v restera collé a cet ensemble de littéraux un grand nombre de fois. Cette intuition est juste une hypothèse, et elle n'est encore appuyée par une validation expérimentale, si ce n'est par les performances de solveur GLUCOSE (voir figure 1).

Série	F_1	F_2	F_3	F_4	F_5	F_6
een	78 (72–83)	83 (73–93)	13 (11–16)	76 (69–83)	81 (71–90)	81 (71–91)
goldb	57 (13–89)	53 (14–93)	26 (1–99)	54 (13–80)	50 (14–66)	40 (11–68)
grieu	62 (42–87)	64 (45–86)	11 (1–17)	59 (41–87)	60 (44–86)	59 (34–84)
hoons	43 (25–77)	46 (26–86)	13 (7–24)	39 (20–77)	44 (23–86)	47 (22–98)
ibm-2002	41 (9–94)	46 (9–93)	11 (0–52)	37 (8–93)	40 (8–91)	40 (4–92)
ibm-2004	52 (12–88)	54 (15–90)	16 (0–90)	47 (9–88)	49 (9–88)	46 (10–90)
manol-pipe	60 (9–98)	62 (11–98)	15 (0–91)	57 (5–98)	58 (2–97)	58 (3–97)
miz	66 (30–85)	65 (40–81)	11 (0–39)	63 (25–83)	62 (30–81)	59 (34–79)
schup	75 (53–94)	77 (54–93)	26 (4–79)	74 (53–94)	75 (54–93)	86 (55–95)
simon	72 (31–97)	73 (30–96)	49 (5–97)	64 (16–97)	63 (19–96)	61 (23–89)
vange	30 (14–38)	32 (14–42)	0 (0–2)	29 (10–38)	31 (11–42)	27 (4–38)
velev	50 (6–87)	50 (7–86)	10 (0–90)	45 (6–82)	44 (3–84)	39 (0–85)
all	56 (6–98)	58 (7–98)	15 (0–99)	52 (5–98)	53 (2–97)	50 (0–135)

TAB. 2 – Moyenne des valeurs de r (multiplié par 100 et arrondi pour plus de lisibilité) calculée sur les diverses lois de régression proposées. Les valeurs entre parenthèses sont les valeurs minimum et maximum obtenues.

3 Les niveaux de décision décroissent de plus en plus lentement

Nous avons donc mis en évidence une relation entre le passé de la recherche (en observant uniquement les niveaux de décision) et le nombre de conflits nécessaires pour terminer cette recherche. À partir de là, nous avons essayé d'aller plus loin, en regardant si, à partir d'un certain nombre de conflits, la justification permettait de prédire la fin de la recherche. C'est-à-dire, si il existe un nombre de conflits après lequel le nombre $-b/a$ se stabilise.

Ceci à échoué – sans surprise pourrait-on dire – au moins avec la régression linéaire. La prédiction calculée se déplace constamment vers la droite tout au long de la recherche. Sur certains problèmes, nous avons néanmoins observé une certaine régularité. La prédiction $p(X)$ faite au conflit X ($p(X)$ est égal à $-b/a$ avec a et b les caractéristiques de la loi de régression calculées sur les X premiers conflits) est reliée à celle faite au conflit $X+1$ de la manière suivante : $p(X+1) = p(X) + e$ avec e dépendant de l'instance.

Ce type d'observation suggère que plus le nombre de conflits augmente, plus la droite de régression s'aplatit. C'est pour cela que nous avons testé si une loi de décroissance exponentielle ne se cache pas derrière la décroissance des niveaux de décision.

3.1 Existe-t-il une loi générale guidant la décroissance des niveaux de décision ?

Même si la justification à posteriori donne de bons résultats (voir figure 2) les valeurs r sont très mauvaises. Nous avons donc essayé de faire correspondre notre nuage de points (numéro du conflit, niveau de décision) avec une large famille de loi de régression. La première difficulté est dûe à la grande variance de notre ensemble de données. Même si une tendance générale des lois de régression semble naturelle dans certains cas, nous devons réduire cette variance pour tous ces nuages de points. Il existe de nombreuses méthodes pour lisser ces fluctuations. Nous proposons d'utiliser la moyenne lissée. Chaque ordonnée de notre nuage de points (c'est le niveau de décision) est la moyenne des ordonnées de 2000 points précédents et suivants. Pour accentuer ce lissage nous répétons cette opération 5 fois.

Une fois le nuage de points lissé, nous pouvons tester les valeurs de Bravais sur plusieurs types de courbes. Le procédé que nous avons utilisé sur les courbes de type $F_1 : y = a \times x^b + c$ (a, b, c sont réels) est le suivant. Nous avons itéré la valeur de b entre -10 et 10 par pas de 0.1 et calculé à chaque fois la régression obtenue (nous avons donc calculé à chaque fois les valeurs de a et c) ainsi que le coefficient r. Nous gardons celle qui donne le meilleur coefficient r. À partir de là, nous pouvons tester d'autres types de courbes par de simples transformations mathématiques. Voici celles que nous avons choisies :

$$
\begin{aligned}
F_2 :&\quad y = exp(a \times x^b + c)\\
F_3 :&\quad y = ln(a \times x^b + c)\\
F_4 :&\quad y = a \times ln(x)^b + c\\
F_5 :&\quad y = exp(a \times ln(x)^b + c)\\
F_6 :&\quad y = a \times exp(x \times b) + c
\end{aligned}
$$

Le tableau 2 donne une idée des différentes valeurs de r obtenues. La première observation que l'on peut faire est que les valeurs des coefficients r obtenues sont meilleures qu'avec une simple régression linéaire. Ceci est du à l'exposant b, mais aussi au lissage effectué sur les courbes (ce n'est pas le cas dans la section 2). Si nous regardons attentivement, nous remarquons que les courbes F_1 et F_2 sont celles qui suivent le mieux nos nuages de points. Les autres fonctions peuvent donner de très bons résultats sur certains types d'instances (comme F_3 sur les séries `gold`), mais

Series	F_1	F_2
een	78 (72–83)	83 (73–93)
goldb	51 (13–89)	53 (14–93)
grieu	62 (42–87)	64 (45–86)
hoons	77 (77–77)	86 (86–86)
ibm-2002	45 (9–94)	48 (9–93)
ibm-2004	50 (12–84)	55 (15–90)
manol-pipe	59 (9–98)	64 (11–98)
miz	69 (56–80)	70 (60–81)
schup	69 (53–83)	77 (54–93)
simon	75 (55–86)	78 (63–89)
vange	14 (14–14)	32 (14–42)
velev	52 (12–80)	57 (7–86)
all	57 (9–98)	60 (7–98)

TAB. 3 – Valeurs de la table 2, en se focalisant uniquement sur les instances exhibant une décroissance des niveaux de décision.

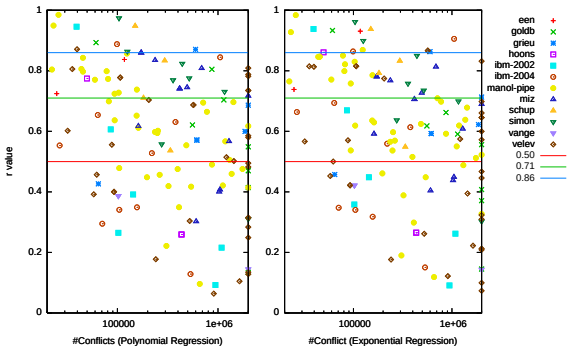


FIG. 3 – Valeurs de r pour deux lois de régressions sélectionnés (gauche : polynômiale (F_1), droite : exponentielle (F_2). Chaque point représente une instance.

leurs moyennes sur toutes les instances est moins bonne que celles des deux premières fonctions. Il est aussi important de noter que de grosses divergences existent sur le coefficient r même au sein d'une même famille.

Nos hypothèses sont basées sur une décroissance des niveaux de décision le long de la recherche, ce qui n'est pas le cas de toutes les instances. La table 3 se concentre sur les instances présentant ces décroissances et sur les régressions de type F_1 et F_2. Les valeurs des coefficients r sont bien meilleures, montrant un meilleur fit sur ces instances. La fonction F_2 semblent donner les meilleurs résultats.

Comme nous venons de le voir, les valeurs de r sont très difficiles à résumer sur un aussi grand nombre d'instances. Nous proposons, dans la figure 3, une vue aérienne de celles-ci, en se focalisant une nouvelle fois sur les régressions de type F_1 et F_2. Chaque point (x, y) correspond à une instance. x correspond au nombre de conflits nécessaires pour la résoudre, et y à la valeur de r. Il est tout d'abord intéressant de noter que les valeurs r semblent devenir de plus en plus mauvaises au fur et à mesure que le nombre de conflits nécessaires pour résoudre l'instance donnée croît (les points semblent se déplacer du haut,gauche vers le bas,droite). Nous pouvons également observer que de nombreuses instances ont un coefficient r relativement bon. Sur les 140 instances testées, la régression polynômiale (de type F_1) a 10 instances avec une valeur de r au dessus de 0.86 et 48 au dessus de 0.71. La régression exponentielle (de type F_2) en a 15 au dessus de 0.86 et 44 au dessus de 0.71.

Arrivés à ce point, il faut bien avouer que le tableau n'est pas très clair. Nous ne sommes pas capable, à partir de données statistiques, de préférer une régression polynômiale ou exponentielle. Néanmoins, et cela est important, il y a très peu de chance qu'une équation de type $y = exp(a \times x^b + c)$ ne puisse pas être approchée par une autre de type $y = a_2 \times x^{b_2} + c_2$, et ceci, lorsque x varie entre 0 et 2 millions et que les données sont très bruitées.

3.2 Courbes et régression

Comme nous venons de le voir, le coefficient r ne nous permet pas de choisir quel type de régression est le plus approprié. Néanmoins, ce coefficient a ses propres limites. En effet, comme nous l'avons montré dans la section 2.3, même de simples lois de régressions linéaires avec un coefficient r très mauvais sont viables dans la pratique. En effet, le coefficient r dépend de la variance des données, et même sur nos nuages de points lissés, les points extrêmes peuvent être très éloignés et avoir une grosse influence dans le calcul de r. Il y a aussi très peu de chances que ce coefficient puisse faire une différence entre deux régressions relativement proches. Pour se convaincre de cela, nous montrons dans la figure 4 les nuages lissés et les lois de régressions calculées sur quelques problèmes sélectionnés parmi les plus représentatifs, avec une préférence sur les instances difficilement résolues (jusqu'à 2 millions de conflits). Nous utilisons $y = exp(a \times x + b)$ comme loi de régression linéaire.

La première courbe `goldb-heqc-i10mul` montre deux caractéristiques intéressantes. Premièrement, après une décroissance régulière durant les 200 000 premiers conflits, il semble que les niveaux de décision restent relativement stables, avec une légère décroissance. Le deuxième point important est le nombre de pics présents sur cette courbe. Ce phénomène est présent sur de nombreux problèmes. Il semble être du à des restarts (soit voulus par le solveur, soit forcés après l'apprentissage d'une clause unaire). Notons tout de même que ce phénomène n'intervient pas systématiquement avec chaque restart. La deuxième courbe `goldb-heqc-i10mul` montre une décroissance qui commence rapidement et qui devient de plus en plus lente avec également quelques pics jusqu'à la fin.

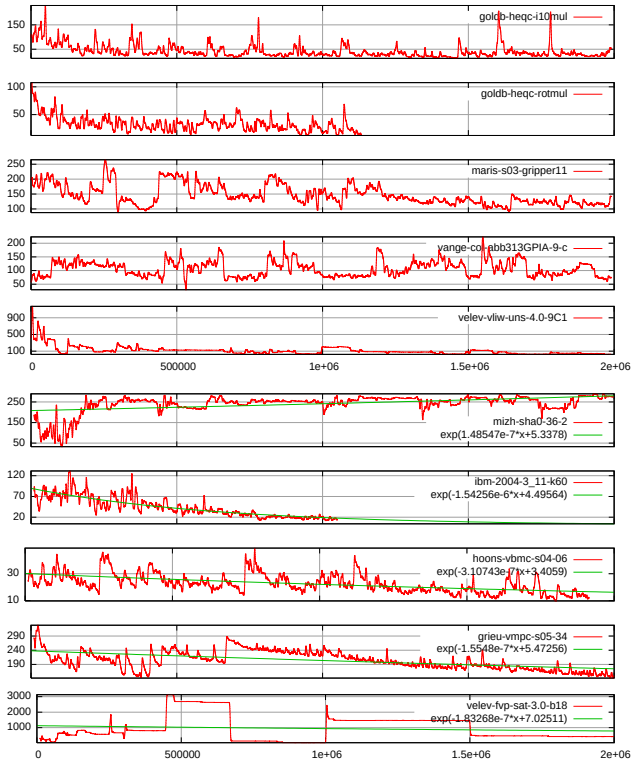


FIG. 4 – Sélection de courbes lissées. Les 5 dernières sont accompagnées de leur loi de régression.

Les courbes `maris` et `vange` montrent des cas typiques avec une très grande variance des données. Enfin, la courbe `velev-vliw` a également une décroissance de plus en plus lente ce qui est, une nouvelle fois, typique.

Les 5 dernières figures montrent, en plus du nuage de points, les lois de régressions calculés. La première (`mizh` montre un cas, relativement rare, de croissance tout au long de la recherche. De telles courbes semblent vraiment dépendre du type de problème encodé (ici des problèmes de cryptographie). La courbe `ibm-2004` est assez incroyable. La loi de régression associée la suit complètement. La courbe `grieu-vmpc-s05-34` est aussi intéressante. Après 700 000 conflits, la courbe se met à décroître régulièrement. Avant cela, une grande variance existe, il est alors difficile d'en déduire une loi de régression. Enfin, la courbe `velev-fvp-sat` montre un cas typique de grandes occiliations. Avoir un coefficient r de bonne qualité semble improbable dans ce cas. Néanmoins, on peut observer que notre hypothèse de départ est ici tout à fait justifiée : les niveaux de décision décroissent toujours et de plus en plus faiblement.

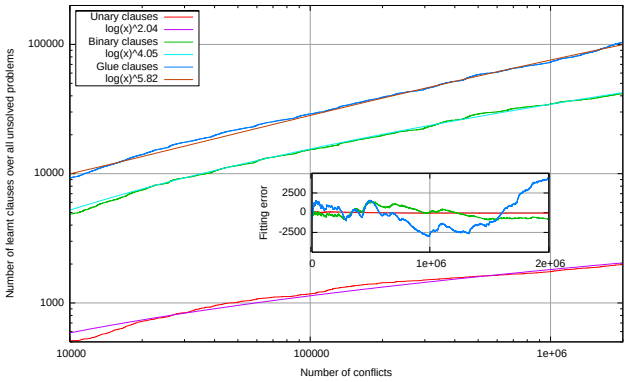


FIG. 5 – Nombre cumulé de clauses apprises (unaires, binaires, glues) produites durant la recherche, restreinte aux problèmes non résolus. En petit, nous avons représenté l'erreur de "fitting".

4 La fréquence des bonnes clauses décroît

La vitesse de décroissance du nombre de décision n'est pas l'unique moyen de mesurer les performances d'un solveur sur une instance donnée. Afin de mieux comprendre les forces et faiblesses des solveurs CDCL, nous avons aussi observé la capacité de ceux-ci de produire des "*bonnes clauses*" durant la recherche. Notre hypothèse est que la décroissance devient de moins en moins rapide parce que le solveur a de plus en plus de mal à générer des clauses intéressantes.

Dans cette section, nous mesurons la *fréquence* d'apparition de certaines clauses, de manière à connaître combien de clauses d'un certain type sont apprises à chaque conflit en moyenne pour l'ensemble des instances. Si la courbe résultante est une droite alors la fréquence d'apprentissage est maintenue tout au long de la recherche. Nous avons donc mesuré les clauses de taille plus petite que 10 et les clauses glues. Nous avons toujours une borne de 2 millions de conflits, mais nous n'avons pris en compte que les instances n'ayant pas été résolues en 2 millions de conflits. Sinon, la fréquence calculée aurait été biaisée. De plus, cela nous permet de nous focaliser sur les instances difficiles.

Les résultats obtenus sont visibles figure 5. On peut noter que la production des clauses unaires et binaires survient de moins en moins au fur et à mesure de la recherche. Nous avons ajouté à cette figure les régression de type $log(x)^b$ calculés sur la clauses unaires et glues. Comme le montre la sous figure interne à la figure 5, le taux d'erreur est très faible, ce qui montre bien que la production cumulée de clauses glues et unaires (c'est la même chose pour les binaires) suit une loi logarithmique : de moins en moins de clauses intéressantes sont produites au fur et à mesure que la recherche avance.

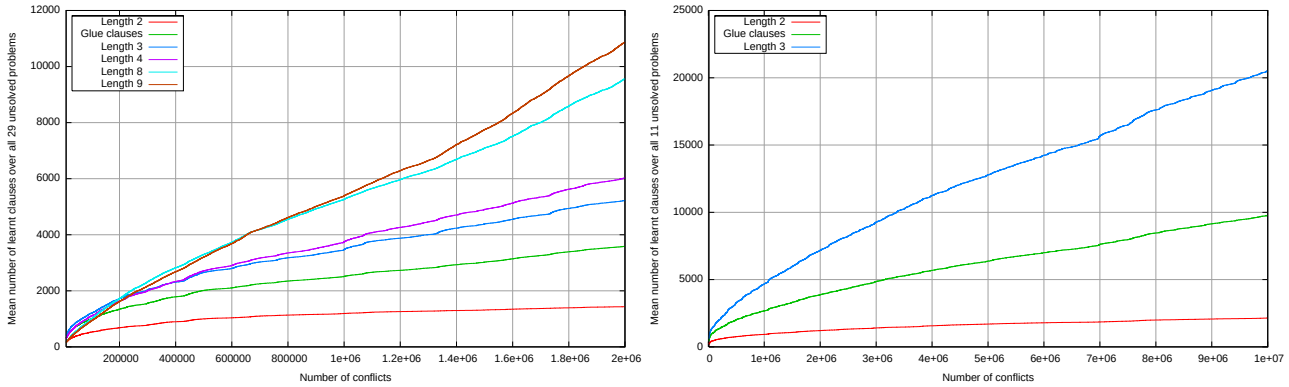


FIG. 6 – Moyenne cumulée de différentes types de clauses apprises avec un arrêt de la recherche à 2 000 000 de conflits (gauche) et 10 000,000 (droite).

La figure 6 de gauche montre le nombre moyen de clauses de différents types appris durant la recherche. La production de clauses glues, unaires et binaires suit une loi logarithmique, ce qui ne semble pas être le cas pour les clauses de plus grande taille. De plus, on produit moins de clauses glues que de clauses ternaires. Pourtant les clauses glues peuvent être de n'importe quelle taille. Pour autant, cette différence semble minime. Pour voir comment cette différence évolue, on a relancé MINISAT, mais cette fois avec un nombre limite de conflits fixé à 10 millions. Il est alors clair que MINISAT a beaucoup de mal à maintenir une bonne production de clauses glues.

Même si cela n'est pas reporté ici, nous avons observé qu'il se passe la même chose entre la génération de deux clauses unaires. Le nombre de clauses glues générés entre la production de deux clauses unaires est de plus en plus rare.

4.1 Obervation de différentes versions de MINISAT

La production de clauses intéressantes décroît donc tout au long de la recherche, pour dédouaner ce résultat des particularités de MINISAT, nous avons réalisé les mêmes expérimentations, mais avec différentes versions de ce solveur. La première version utilise des redémarrages très rapides (tous les 250 conflits), une avec un vardecay de 0.85 au lieu de 0.95 (cette constante contrôle la vitesse à laquelle le solveur oublie les variables lors du choix heuristique), et enfin une avec la sauvegarde de la phase [12] et des redémarrages à la luby [6]. Ici, une limite de 2 millions de conflits est à nouveau utilisée.

Les résultats sont visibles sur la figure 7. Il est frappant de noter que la version avec luby et sauvegarde de la phase produit peu de clauses unaires, mais beaucoup de clauses binaires et glues. Si on regarde attentivement, on peut également remarquer que la différence entre cette version et avec un restart250 n'est pas si importante. Il semble que la version phase+luby soit meilleure pour produire des clauses binaires, et que la production de clauses glues paie vraiment. Nous pouvons aussi noter que la version vardecay est meilleure que MINISAT dans la production de clauses glues, même si elle donne au final de plus mauvais résultats (voir la sous figure cactus). La notion de clauses glues n'est donc pas suffisante pour expliquer le bon ou le mauvais comportement des différentes versions. Nous pensons que ces clauses sont importantes mais il reste encore du travail pour appréhender totalement leur rôle dans les démonstrateurs CDCL.

5 Conclusion

Dans cet article, nous avons considéré le solveur MINISAT comme un système physique a étudier. Nous nous sommes focalisés sur deux points qui nous semblent importants : les niveaux de décision et certaines caractéristiques des clauses apprises durant la recherche (taille et nombre de niveaux de décision). A partir de là, nous avons construit un certain nombre d'expériences pour valider ou invalider certaines hypothèses.

Ces premiers résultats tendent à montrer que dans de nombreux cas, MINISAT réduit, au fur et à mesure que la recherche avance, le nombre de niveaux de décision nécessaire pour atteindre un conflit. Néanmoins, au plus cette recherche avance, au plus il semble difficile de réduire ces niveaux de décision. Nous suspectons que ces niveaux de décision suivent une courbe exponentielle qui expliquerai pourquoi les solveurs CDCL présentent des figures cactus. De plus, nous montrons, que plus la recherche avance, moins les clauses apprises semblent intéressantes.

Références

[1] G. Audemard and L. Simon. Predicting learnt clauses quality in modern sat solvers. In *proceedings of IJ-*

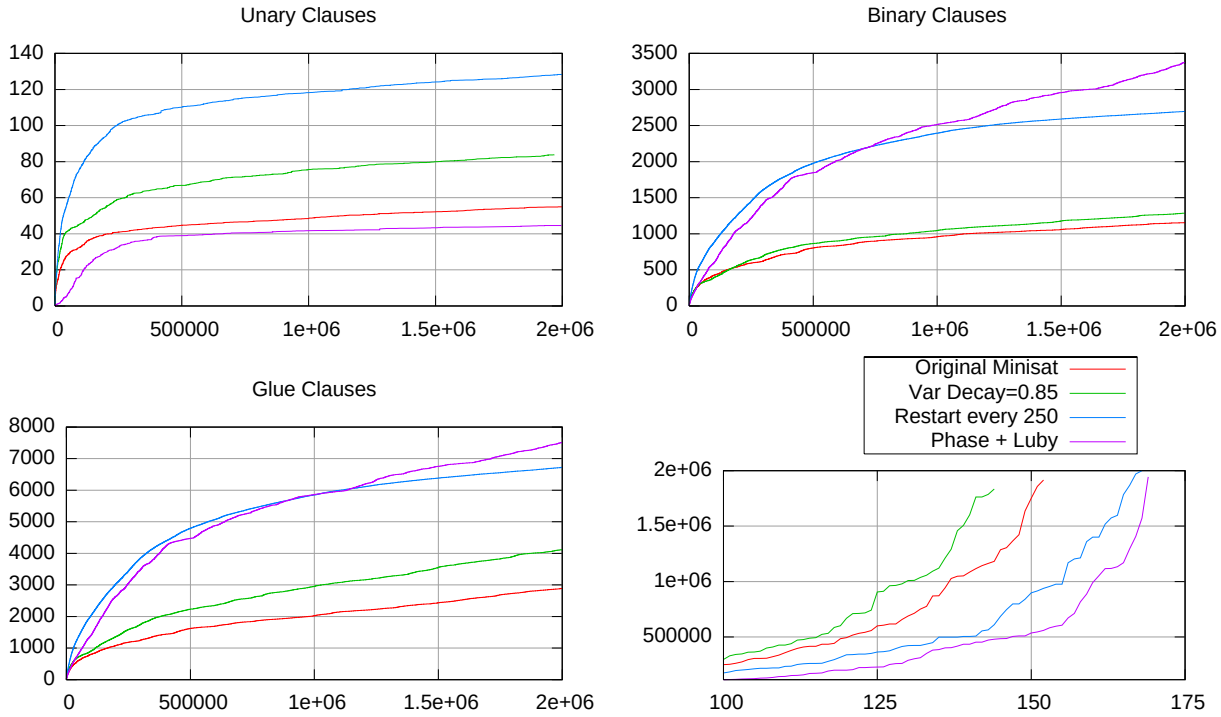


FIG. 7 – Fréquence de production de certains types de clauses pour les quatre versions de MINISAT. La figure en bas à gauche donne la figure de type cactus, du point de vue des conflits (sur tous les problèmes résolus).

CAI, 2009. to appear.

[2] A. Biere. PicoSAT essentials. *Journal on Satisfiability*, 4 :75–97, 2008.

[3] N. Eén and N. Sörensson. An extensible SAT-solver. In *proceedings of SAT*, pages 502–518, 2003.

[4] C. Gomes, B. Selman, N. Crato, and H. Kautz. Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. *Journal of Automated Reasoning*, 24 :67–100, 2000.

[5] J. N. Hooker. Needed : An empirical science of algorithms. *Operations Research*, 42 :201–212, 1994.

[6] J. Huang. The effect of restarts on the efficiency of clause learning. In *proceedings of IJCAI*, pages 2318–2323, 2007.

[7] S. Jabbour and L. Sais. personnal communication, February 2008.

[8] Said Jabbour. *De la satisfiabilité propositionnelle aux formules booléennes quantifiées*. PhD thesis, Université d'Artois, 2008.

[9] Chu Min Li and Anbulagan. Look-ahead versus look-back for satisfiability problems. In *Principles and Practice of Constraint Programming (CP97)*, volume 1330 of *LNCS*, pages 341–355. Springer Verlag, 1997.

[10] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff : Engineering an efficient SAT solver. In *proceedings of DAC*, pages 530–535, 2001.

[11] J. Peter and R. Hull. *Le principe de Peter*. 1969.

[12] K. Pipatsrisawat and A. Darwiche. A lightweight component caching scheme for satisfiability solvers. In *proceedings of SAT*, pages 294–299, 2007.

[13] M. Prasad, A. Biere, and A. Gupta. A survey of recent advances in SAT-based formal verification. *journal on Software Tools for Technology Transfer*, 7(2) :156–173, 2005.

[14] João P. Marques Silva and Karem A. Sakallah. Grasp - a new search algorithm for satisfiability. In *proceedings of ICCAD*, pages 220–227, 1996.

[15] R. Williams, C. Gomes, and B. Selman. Backdoors to typical case complexity. In *Proceedings of IJCAI'03*, 2003.

[16] L. Zhang and C. Madigan. Efficient conflict driven learning in a boolean satisfiability solver. In *proceedings of ICCAD*, pages 279–285, 2001.

[17] Lintao Zhang and Sharad Malik. Cache performance of sat solvers : a case study for efficient implementation of algorithms. In *proceedings of SAT*, pages 287–298, 2003.

Décidabilité de contraintes du premier ordre par contraintes duales *

Khalil Djelloul

Laboratoire d'Informatique Fondamentale d'Orléans.
Bat. 3IA, rue Léonard de Vinci. 45067 Orléans, France.
khalil.djelloul@univ-orleans.fr

Résumé

Satisfiability modulo theories (SMT) est une discipline issue de la vérification formelle qui a de nombreuses retombées pratiques en programmation par contrainte et intelligence artificielle. L'idée est simple : ayant une théorie T modélisée par un ensemble d'axiomes, est-il possible de générer une procédure de décision qui calcul la valeur de vérité dans T de toute contrainte du premier ordre sans variables libres ? Un des derniers résultats publiés est la décomposabilité : une propriété partagée par plusieurs théories du premier ordre qui nous a permis de dégager une procédure de décision générique pour toute théorie décomposable. Cette dernière utilise une première phase de calcul qui transforme la formule initiale en une formule dite normalisée. Nous discutons dans ce papier les avantages et inconvénients d'une telle transformation et proposons une nouvelle procédure de décision qui n'a pas recours à la normalisation.

Abstract

Satisfiability modulo theories (SMT) is a discipline that comes from formal verification and has many practical applications in constraint programming and artificial intelligence. The idea is simple : having a theory T in the form of a set of axioms, is it possible to produce a decision procedure that computes the truth value in T for any first-order constraint without free variables ? One of the last published results concerns decomposability : a new property shared by several first-order theories which allowed us to build a general decision procedure for any decomposable theory. This latter uses a pre-processing step which transforms the initial formula into a so-called normalized one. We discuss in this paper the advantages and disadvantages of such a transformation and propose a new decision procedure which does not use such a normalization.

*Une version complète de cet article vient d'être acceptée pour paraître dans *Recent advances in constraints*, numéro spécial de LNAI : F. Rossi et F. Fages (Eds). A paraître.

1 Introduction

La programmation par contraintes (PPC) est une discipline au carrefour de l'intelligence artificielle, de la recherche opérationnelle et de l'analyse numérique, qui est née dans les années 70 et qui a pour ambition de résoudre n'importe quel problème combinatoire, y compris des problèmes classiques de recherche opérationnelle [10]. De nos jours, la PPC est en plein essor industriel et est utilisée avec succès dans des domaines d'applications réellement diversifiés tels que l'ordonnancement, l'analyse financière, la simulation et la synthèse de circuits intégrés.

En premier lieu, les chercheurs se sont tout d'abord intéressés à la résolution efficace des problèmes de satisfaction de contraintes (CSP) : un ensemble de contraintes qui doivent être satisfaites par des éléments de différents domaines. L'utilisation de quantificateurs n'était bien entendu pas à l'ordre du jours vu qu'elle faisait passer le problème de la classe NP-complet à PSPACE complet. Il a fallut attendre les années 2000 pour trouver une communauté de chercheurs qui s'intéresse aux QCSP : des problèmes de satisfaction de contraintes quantifiées. En effet, en cette période, on disposait d'un grand nombre de résultats théoriques et pratiques qui permettait de résoudre des instances de CSP que l'on imaginait jamais aborder il y a une dizaine d'années. Il devenait alors intéressent d'étudier des extensions de ces outils du cadre CSP à celui des QCSP ; et c'est ainsi qu'un on vu aboutir plusieurs travaux autour de la quantification [4, 2, 3]. Programmer et modéliser avec des quantificateurs n'est donc plus une idée si utopique que ça ! Pourquoi donc ne pas aller encore plus loin et aborder la résolution de contrainte du premier ordre !

Ainsi, des chercheurs issues de la communauté de la

vérification formelle et de la PPC se sont alliés pour étudier la classe des problèmes SMT : *Satisfiability Modulo Theories* [12]. Le problème est simple : ayant un ensemble d'axiomes ϕ sur une signature contenant un ensemble de fonctions F et un ensemble de relations R, peut on déduire une procédure de décision dans la théorie résultante ? Ce genre de problème est très utile en vérification (optimisation de compilateur, vérification de propriétés dans un système dynamique,...etc). Une bonne synthèse à ce sujet est disponible dans [1].

Dans ce cadre la, nous avons proposé en 2007 une condition suffisante - dite de décomposabilité - pour qu'une théorie soit complète et admette une procédure de décision sous forme de quelques règles de réécritures [8]. Nous avons alors montré que plusieurs théories satisfaisaient cette condition et avons publié récemment une version plus élaborée de cette procédure de décision [6]. L'idée de base était simple : au lieu de manipuler des formules qui peuvent contenir différents symboles logiques, il vaut mieux les transformer d'abord en formules contenant uniquement les symboles $\{\exists, \wedge, \neg\}$ puis exploiter les propriétés de la décomposabilité. On a alors généralisé le concept de formules normalisées (FN) - introduit par Dao dans le cadre de la théorie des arbres [5] - et l'avons utilisé en amont de notre procédure de décision. Le schéma de décision d'une proposition φ était donc le suivant : transformer φ en une formule normalisée puis appliquer la décomposabilité jusqu'à aboutir à vrai ou faux.

Observons maintenant de plus prés les formules normalisées : ce sont des formules de la forme

$$\neg(\exists x_1 ... \exists x_n \, (\alpha \wedge \bigwedge_{i \in I} \varphi_i)), \tag{1}$$

où α est une conjonction de formules atomiques et les φ_i des sous-formules de la même forme que (1). Choisissons par exemple une théorie T de signature contenant les deux fonctions unaires f et g. La formule suivante est normalisée :

$$\neg \left[\exists y \, y = f(y) \wedge \left[\begin{array}{l} \neg(\exists x \, y = f(x) \wedge x = g(y) \wedge \neg(\exists z \, z = g(y))) \wedge \\ \neg(\exists v \, v = f(y)) \end{array} \right] \right].$$

Le problème que nous avons constaté est que dans la plus part des cas, la formule normalisée φ obtenue à partir de la formule ϕ est beaucoup plus grande et complexe que φ (beaucoup plus d'alternations de quantificateurs et de négations). En effet, si l'on considère la théorie P de Presburger [9] et si φ est la formule

$$\forall x \exists y \, y \neq x,$$

alors ϕ est la formule normalisée suivante

$$\neg(\exists \, x \; vrai \wedge \neg(\exists y \; vrai \wedge \neg(\exists \epsilon \, y = x)),$$

où $\exists \epsilon$ est la quantification vide. Dans cette exemple, nous sommes passés d'une formule très simple à une formule contenant trois quantificateurs imbriqués avec des négations. En utilisant alors notre procédure de décision [6] sur ϕ, nous obtenons un temps d'exécution qui est beaucoup plus grand que celui obtenu par simplification de φ en *vrai* via l'axiome qui affirme que pour tout élément x il existe un y qui est différent dans tous les modèles de P.

Nous avons alors essayer de comprendre cette perte de temps d'exécution entre les deux approches. L'explication est la suivante : a chaque fois que notre procédure de décision identifie dans une sous formule normalisée deux quantificateurs imbriqués avec une négation, elle applique une règle très coûteuse en temps et espace pour supprimer cette imbrication aux prix d'augmenter exponentiellement la taille de la formule résultante. En d'autres termes, plus on a d'alternation de quantificateurs et de négations, plus le temps d'exécution sera long. Il serait donc plus intéressant pour nous si l'on pouvait produire une procédure de décision qui n'a pas recours à la normalisation de la formule.

Contributions : Dans ce papier, nous proposons une nouvelle procédure de décision pour les théories fonctionnelles décomposables, c'est-à-dire les théories décomposables dont la signature ne contient pas de symboles de relation autres que $=$ et $\neq$ [1]. Notre algorithme n'utilise pas de formules normalisées et s'exécute directement en une seule passe sur la formule initiale. Il utilise de nouvelles propriétés issues de la décomposabilité et s'exprime par un ensemble de règles de réécriture qui, une fois un point fixe atteint, produit une combinaison booléenne de formules de base qui peuvent être immédiatement réduites à vrai ou à faux dans la théorie fonctionnelle considérée.

Ce papier est organisé en trois sections suivies par une conclusion. Cette introduction constitue la première section. La section 2 est consacrée à un rappel de la logique du premier ordre et des théories décomposables. Nous présentons dans la section 3 la notion de *contraintes duales* et *contraintes de base*, et proposons un algorithme de décision sous forme d'un ensemble de règles de réécriture qui manipule des contraintes duales et produit une combinaison booléenne de contraintes de base. Nous terminons ce papier par une longue discussion sur les avantages et inconvénients de la phase de normalisation par rapport à notre procédure de décision. Nous parlons également de performances de nos algorithmes et des travaux futurs autour de la décomposabilité et de la résolution de contraintes du premier

1. Bien entendu, ces théories peuvent avoir un ensemble de fonctions quelconque.

ordre dans le cadre des SMT.

2 Préliminaires

2.1 Contrainte du premier ordre

Soit V un ensemble infini de variables. Soit S un ensemble de symboles, appelé signature, et partionné en deux ensembles disjoints : l'ensemble F des symboles de fonction et l'ensemble R des symboles de relation. A chaque symbole de fonction et relation est associé un entier strictement positif n appelé *arité*. Un symbole n-air est un symbole d'arité n. Une formule ou contrainte du premier ordre est une expression de l'une des onze formes suivantes :

$$\begin{array}{c} s = t,\ r(t_1, \ldots, t_n),\ \mathit{vrai},\ \mathit{faux}, \\ \neg\varphi,\ (\varphi \wedge \psi),\ (\varphi \vee \psi),\ (\varphi \rightarrow \psi),\ (\varphi \leftrightarrow \psi), \\ (\forall x\, \varphi),\ (\exists x\, \varphi), \end{array} \quad (2)$$

avec $x \in V$, r un symbole de relation n-aire pris dans R, φ et ψ des formules plus petites, s, t et les t_i des termes, c'est-à-dire des expressions de l'une des deux formes suivantes : $x, f(t_1, ..., t_n)$, avec x pris dans V, f un symbole de fonction n-aire pris dans F et les t_i des termes plus courts. Les formules de la première ligne de (2) sont dites *atomiques*, et *à plat* si elles sont de l'une des formes suivantes :

$$\mathit{vrai},\ \mathit{faux},\ x_0 = x_1, x_0 = f(x_1, ..., x_n),\ r(x_1, ..., x_n),$$

où les x_i sont des variables (éventuellement non distinctes) prises dans V, $f \in F$ et $r \in R$. Notons AT l'ensemble des conjonctions de formules atomiques à plat.

Une occurrence d'une variable x dans une formule est dite *liée* si elle apparaît dans une sous formule de la forme $(\forall x\, \varphi)$ ou $(\exists x\, \varphi)$. Elle est *libre* dans tous les autres cas. Les *variables libres* sont celles qui ont au moins une occurrence libre dans cette formule. Une *proposition* est une formule sans variables libres.

Un *modèle* est un couple $M = (D, F)$, où D est un ensemble non vide d'individus de M et F un ensemble de fonctions et de relations dans D. On appelle *instanciation* d'une formule φ par des individus de M, la formule obtenue à partir de φ en remplaçant chaque variable libre x de φ par le même individu i de D et en considérant chaque élément de D comme un symbole de fonction d'arité 0.

Une *théorie* T est un ensemble de propositions éventuellement infini. On dit que le modèle M est un *modèle de* T, si pour tout élément φ de T, $M \models \varphi$. Si φ est une formule, on écrit $T \models \varphi$ si pour tout modèle M de T, $M \models \varphi$. Une théorie T est *complète* si pour toute proposition φ, une et une seule des propriétés suivantes est satisfaite : $T \models \varphi$, $T \models \neg\varphi$.

2.2 Vecteur quantifié

Soit M un modèle et T une théorie. Soit $\bar{x} = x_1 \ldots x_n$ et $\bar{y} = y_1 \ldots y_n$ deux mots de V de même longueur. Soit φ, et $\varphi(\bar{x})$ deux formules. On note

$\exists\bar{x}\, \varphi$ pour $\exists x_1 ... \exists x_n\, \varphi$,
$\forall\bar{x}\, \varphi$ pour $\forall x_1 ... \forall x_n\, \varphi$,
$\exists?\bar{x}\, \varphi(\bar{x})$ pour $\forall\bar{x}\forall\bar{y}\, \varphi(\bar{x}) \wedge \varphi(\bar{y}) \rightarrow \bigwedge_{i\in\{1,\ldots,n\}} x_i = y_i$,
$\exists!\bar{x}\, \varphi$ pour $(\exists\bar{x}\, \varphi) \wedge (\exists?\bar{x}\, \varphi)$.

Le mot $\bar{x}$, qui peut être le mot vide ε, est appelé *vecteur de variables*. Sémantiquement, les deux quantificateurs $\exists?$ et $\exists!$ signifient "au plus un" et "un et un seul".

Introduisons maintenant une notation commode concernant la priorité des quantificateurs : $\exists$, $\exists!$, $\exists?$ et $\forall$.

Notation 2.2.1 *Soit Q un quantificateur pris dans $\{\forall, \exists, \exists!, \exists?\}$. Soit $\bar{x}$ un vecteur de variables pris dans V. On écrit :*

$$Q\bar{x}\, \varphi \wedge \phi \ \ \mathit{pour}\ \ Q\bar{x}\, (\varphi \wedge \phi).$$

Exemple 2.2.2 *Soit $I = \{1, ..., n\}$ un ensemble fini. Soient φ et ϕ_i avec $i \in I$ des formules. Soient $\bar{x}$ et $\bar{y}_i$ avec $i \in I$ des vecteurs de variables. On écrit :*

$\exists\bar{x}\, \varphi \wedge \neg\phi_1$ *pour* $\exists\bar{x}\, (\varphi \wedge \neg\phi_1)$,
$\forall\bar{x}\, \varphi \wedge \phi_1$ *pour* $\forall\bar{x}\, (\varphi \wedge \phi_1)$,
$\exists!\bar{x}\, \varphi \wedge \bigwedge_{i\in I}(\exists\bar{y}_i \phi_i)$ *pour* $\exists!\bar{x}\, (\varphi \wedge (\exists\bar{y}_1\phi_1) \wedge ... \wedge (\exists\bar{y}_n\phi_n) \wedge \mathit{vrai})$,
$\exists?\bar{x}\, \varphi \wedge \bigwedge_{i\in I} \neg(\exists\bar{y}_i \phi_i)$ *pour* $\exists?\bar{x}\, (\varphi \wedge (\neg(\exists\bar{y}_1\phi_1)) \wedge ... \wedge (\neg(\exists\bar{y}_n\phi_n)) \wedge \mathit{vrai})$.

Terminons cette sous-section par deux propriétés qui vont nous êtres utiles pour justifier la correction de nos règles de réécriture.

Propriété 2.2.3 *Si $T \models \exists?\bar{x}\, \varphi$ alors*

$$T \models (\exists\bar{x}\, \varphi \wedge \bigwedge_{i\in I} \neg\phi_i) \leftrightarrow ((\exists\bar{x}\varphi) \wedge \bigwedge_{i\in I} \neg(\exists\bar{x}\, \varphi \wedge \phi_i)).$$

Propriété 2.2.4 *Si $T \models \exists!\bar{x}\, \varphi$ alors*

$$T \models (\exists\bar{x}\, \varphi \wedge \bigwedge_{i\in I} \neg\phi_i) \leftrightarrow \bigwedge_{i\in I} \neg(\exists\bar{x}\, \varphi \wedge \phi_i).$$

2.3 Le quantificateur infini $\exists_\infty^{\Psi(u)}$

Nous rappelons ici la définition du quantificateur infini que nous avons introduit dans [8] et utilisé pour formaliser la propriété de décomposition.

Définition 2.3.1 *[8] Soit M un modèle, $\varphi(x)$ une formule et $\Psi(u)$ un ensemble de formules ayant au plus une seule variable libre u. On écrit*

$$M \models \exists_\infty^{\Psi(u)} x\, \varphi(x), \quad (3)$$

si pour toute instanciation $\exists x\,\varphi'(x)$ *de* $\exists x\,\varphi(x)$ *par des individus de* M *et pour tout sous-ensemble fini* $\{\psi_1(u),..,\psi_n(u)\}$ *d'éléments de* $\Psi(u)$*, l'ensemble des individus* i *de* M *tel que* $M \models \varphi'(i) \wedge \bigwedge_{j\in\{1,...,n\}} \neg\psi_j(i)$ *est infini.*

Notons que si $\Psi(u) = \{faux\}$ alors (3) exprime simplement le fait que M contienne un ensemble infini d'individus i tel que $\varphi(i)$ est vraie. Informellement, la notation (3) signifie qu'il existe une élimination complète de quantificateurs sur les formules de la forme $\exists x\,\varphi(x) \wedge \bigwedge_{j\in\{1,...,n\}} \neg\psi_j(x)$ du fait qu'il existe une infinité d'individus x du modèle M qui satisfont cette formule.

Propriété 2.3.2 *[8] Soit* J *un ensemble fini (éventuellement vide). Soit* $\varphi(x)$ *et* $\varphi_j(x)$*, avec* $j \in J$*, des* M*-formules. Si* $T \models \exists_\infty^{\Psi(u)} x\,\varphi(x)$ *et si pour tout* $\varphi_j(x)$*, au moins une des propriétés suivantes est satisfaite*

- $T \models \exists? x\,\varphi_j(x)$,
- *il existe* $\psi_j(u) \in \Psi(u)$ *tel que* $T \models \forall x\,\varphi_j(x) \rightarrow \psi_j(x)$,

alors

$$T \models \exists x\,\varphi(x) \wedge \bigwedge_{j\in J} \neg\varphi_j(x)$$

Propriété 2.3.3 *[8] Si* $T \models \exists_\infty^{\Psi(u)} x\,\varphi(x)$ *alors* $T \models \exists_\infty^{\Psi(u)} x$ *vrai.*

2.4 Théories décomposables

Nous rappelons maintenant la définition des *théories decomposables* [8]. Informellement, cette définition exprime le fait que pour tout théorie décomposable T et pour toute formule de la forme $\exists\bar{x}\,\alpha$, avec $\alpha \in AT$, on peut se ramener à une formule décomposée équivalente dans T de la forme $\exists\bar{x}'\,\alpha' \wedge (\exists\bar{x}''\,\alpha'' \wedge (\exists\bar{x}'''\,\alpha'''))$ où les formules $\exists\bar{x}'\,\alpha'$, $\exists\bar{x}''\,\alpha''$, et $\exists\bar{x}'''\,\alpha'''$ ont des propriétés qui s'expriment à l'aide des quantificateurs $\exists?$, $\exists!$ et $\exists_\infty^{\Psi(u)}$. L'intérêt de la décomposabilité est qu'au lieu de s'attarder à construire l'intégralité d'une procédure de décision sur une théorie T, il suffit juste de vérifier que T est décomposable ; et si tel est le cas alors nous disposons d'une procédure de décision générale.

Il est important de noter que décomposabilité ne rime pas avec "élimination complète de quantificateurs". En effet, nous avons présenté plusieurs théories décomposables qui n'admettent pas d'élimination complète de quantificateurs. La procédure de décision que nous allons présenter dans la section suivante n'est donc pas un simple algorithme d'élimination complète de quantificateurs.

Dans tout ce qui suit nous utilisons l'abréviation "svls" pour *"sans variables libres supplémentaires"*. Une formule φ est équivalente à une formule svls ψ dans T signifie que $T \models \varphi \leftrightarrow \psi$ et que ψ ne contient pas de variables libres autres que celle de φ.

Définition 2.4.1 *Une théorie* T *est dite* decomposable *s'il existe un ensemble* $\Psi(u)$ *de formules, ayant au plus une variable libre* u*, et trois ensemble de formules* A'*,* A'' *et* A''' *de la forme* $\exists\bar{x}\,\alpha$ *avec* $\alpha \in AT$ *tels que*

1. *Toute formule de la forme* $\exists\bar{x}\,\alpha \wedge \psi$*, avec* $\alpha \in AT$ *et* ψ *une formule quelconque, est équivalente dans* T *à une formule svls décomposée de la forme*

   $$\exists\bar{x}'\,\alpha' \wedge (\exists\bar{x}''\,\alpha'' \wedge (\exists\bar{x}'''\,\alpha''' \wedge \psi)),$$

   avec $\exists\bar{x}'\,\alpha' \in A'$*,* $\exists\bar{x}''\,\alpha'' \in A''$ *et* $\exists\bar{x}'''\,\alpha''' \in A'''$*.*
2. *Si* $\exists\bar{x}'\alpha' \in A'$ *alors* $T \models \exists?\bar{x}'\,\alpha'$ *et pour toute variable libre* y *dans* $\exists\bar{x}'\alpha'$*, au moins une des propriétés suivantes est satisfaite :*
   - $T \models \exists? y\bar{x}'\,\alpha'$,
   - *il existe* $\psi(u) \in \Psi(u)$ *tel que* $T \models \forall y\,(\exists\bar{x}'\,\alpha') \rightarrow \psi(y)$*.*
3. *Si* $\exists\bar{x}''\alpha'' \in A''$ *alors pour chaque* x_i'' *de* $\bar{x}''$ *on a* $T \models \exists_\infty^{\Psi(u)} x_i''\,\alpha''$*.*
4. *Si* $\exists\bar{x}'''\alpha''' \in A'''$ *alors* $T \models \exists!\bar{x}'''\,\alpha'''$*.*
5. *Si la formule* $\exists\bar{x}'\alpha'$ *appartient à* A' *et n'a pas de variables libres alors cette formule est soit la formule* $\exists\varepsilon vrai$ *soit la formule* $\exists\varepsilon faux$*.*

Nous avons donné dans [8] puis dans [6] une procédure de décision qui transforme d'abord la proposition initiale en formule normalisée. Nous allons voir dans ce qui suit les inconvénients d'une telle transformation.

2.5 Formules normalisées

Définition 2.5.1 *Une formule normalisée* φ *de profondeur* $d \geq 1$ *est une formule de la forme*

$$\neg(\exists\bar{x}\,\alpha \wedge \bigwedge_{i\in I} \varphi_i),$$

avec I *un ensemble fini (éventuellement vide),* $\alpha \in AT$ *et les* φ_i *des formules normalisées de profondeur* d_i *avec* $d = 1 + \max\{0, d_1, ..., d_n\}$*.*

Exemple 2.5.2 *Soit* φ *la formule suivante*

$$\forall x \exists y\; y \neq x \wedge x = f(x), \tag{4}$$

où f *est un symbole de fonction d'arité 1. La formule précédente est équivalente dans toute théorie* T *à la formule normalisée* ϕ *(de profondeur 3) suivante :*

$$\neg(\exists\, x\; vrai \wedge \neg(\exists y\; vrai \wedge \neg(\exists\, \varepsilon\, y = x \vee x \neq f(x))). \tag{5}$$

Afin de résoudre ϕ*, les procédures de décisions des théories décomposables [8, 6] utilisent une règle qui*

transforme tout formule normalisée de profondeur 3 en une conjonction de formules normalisées de profondeur 2. Cependant, à chaque fois que cette règle diminue la profondeur de la formule, elle augmente exponentiellement la taille de la formule résultante. En effet, nous avons montré dans [6] que cette règle est l'unique responsable de la complexité exponentielle en temps et espace de nos procédures de décision.

D'autre part, la transformation de la formule φ (c'est la formule (4)) en une formule normalisée ϕ (c'est la formule (5)) implique la création de trois quantificateurs imbriqués par négation. Ainsi, nous devons appliquer deux fois de suite la règle très coûteuse de diminution de profondeur si l'on veut arriver à vrai ou à faux. Toutes ces étapes, très coûteuse en temps et espace, peuvent être évitées en utilisant de nouvelles propriétés de décomposabilité et en évitant l'utilisation de formules normalisées.

3 Une procédure de décision pour les théories fonctionnelles décomposables

Nous présentons dans cette section une procédure de décision pour toute théorie fonctionnelle décomposable, c'est-à-dire, toute théorie décomposable dont l'ensemble de relation est réduit à $\{=, \neq\}$.

3.1 Formules duales

Soit T une théorie fonctionnelle décomposable munie de la signature $F \cup \{=, \neq\}$ où F est un ensemble de fonction (éventuellement vide ou infini). Les ensembles $\Psi(u)$, A', A'' et A''' sont donc connus et fixés pour tout le reste de ce papier.

Définition 3.1.1 *Une* formule positive *est une formule qui ne contient pas d'occurrence du symbole $\neg$.*

Il est évident que pour tout théorie fonctionnelle décomposable T, nous pouvons transformer toute formule φ en une formule positive. Pour cela, il suffit de distribuer la négation sur les sous-formules et d'utiliser les lois classiques de la logique du premier ordre.

Exemple 3.1.2 *Soit φ la formule suivante*

$$\exists u_2 \forall u_1 \exists u_3 \neg \left[\begin{array}{l} \exists v_1\, v_1 = f(u_1, u_2) \wedge u_2 = g(u_1) \wedge \\ \neg(\exists w_1\, v_1 = g(w_1)) \wedge \\ \neg(\exists w_2\, u_2 = g(w_2) \wedge w_2 = g(u_3)) \end{array} \right]. \quad (6)$$

La formule précédente est équivalente à la formule positive suivante :

$$\exists u_2 \forall u_1 \exists u_3 \left[\begin{array}{l} \forall v_1\, v_1 \neq f(u_1, u_2) \vee u_2 \neq g(u_1) \vee \\ (\exists w_1\, v_1 = g(w_1)) \vee \\ (\exists w_2\, u_2 = g(w_2) \wedge w_2 = g(u_3)) \end{array} \right]. \quad (7)$$

Définition 3.1.3 *La formule* duale $\overline{\varphi}$ *d'une formule positive φ, est la formule obtenue en remplaçant chaque occurrence de $=$, $\neq$, $\wedge$, $\vee$, $\exists$, $\forall$ par $\neq$, $=$, $\vee$, $\wedge$, $\forall$, $\exists$.*

Exemple 3.1.4 *La formule duale de la formule positive (7) est la suivante :*

$$\forall u_2 \exists u_1 \forall u_3 \left[\begin{array}{l} \exists v_1\, v_1 = f(u_1, u_2) \wedge u_2 = g(u_1) \wedge \\ (\forall w_1\, v_1 \neq g(w_1)) \wedge \\ (\forall w_2\, u_2 \neq g(w_2) \vee w_2 \neq g(u_3)) \end{array} \right].$$

Nous montrons facilement la propriété suivante :

Propriété 3.1.5 *$T \models \varphi \leftrightarrow \overline{(\overline{\varphi})}$ et $T \models \varphi \leftrightarrow \neg\overline{\varphi}$.*

3.2 Formules de base

Définition 3.2.1 *Une formule de base est une formule de l'une des deux formes suivantes :*

$$(\exists \bar{x}\, \alpha), (\forall \bar{x}\, \overline{\alpha}),$$

avec $\alpha \in AT$ et $\bar{x}$ un vecteur de variables (éventuellement vide).

En utilisant la définition 2.4.1, nous montrons la propriété suivante :

Propriété 3.2.2 *Si φ est une formule de base sans variables libres alors elle est de l'une des formes suivantes*

$$(\exists \varepsilon\ vrai),\ (\exists \varepsilon\, faux),\ (\forall \varepsilon\ vrai), (\forall \varepsilon\, faux).$$

Preuve Soit $\exists \bar{x}\, \alpha$ une formule de base avec $\alpha \in AT$. D'après la définition 2.4.1, cette formule est équivalente dans T à une formule svls de la forme

$$\exists \bar{x}'\, \alpha' \wedge (\exists \bar{x}''\, \alpha'' \wedge (\exists \bar{x}'''\, \alpha''')),$$

avec $\exists \bar{x}'\, \alpha' \in A'$, $\exists \bar{x}''\, \alpha'' \in A''$ et $\exists \bar{x}'''\, \alpha''' \in A'''$. Du fait que $\exists \bar{x}'''\, \alpha''' \in A'''$ alors d'après la définition 2.4.1, nous avons $T \models \exists! \bar{x}''' \alpha'''$, donc $T \models \exists \bar{x}''' \alpha'''$. La formule précédente est donc équivalente dans T à

$$\exists \bar{x}'\, \alpha' \wedge (\exists \bar{x}''\, \alpha''),$$

qui est équivalente dans T à

$$\exists \bar{x}'\, \alpha' \wedge (\exists x_1'' ... x_{n-1}''\, (\exists x_n''\, \alpha'')).$$

Du fait que $\exists \bar{x}''\, \alpha'' \in A''$ alors d'après la définition 2.4.1 on a $T \models \exists_\infty^{\Psi(u)} x_n''\, \alpha''$ et donc d'après la propriété 2.3.2 (avec $J = \emptyset$) $T \models \exists\, x_n''\, \alpha''$. Par conséquent, la formule précédente est équivalente dans T à

$$\exists \bar{x}'\, \alpha' \wedge (\exists x_1'' ... x_{n-1}''\ vrai),$$

qui est finalement équivalent dans T à

$$\exists \bar{x}'\, \alpha'.$$

D'après le cinquième point de la définition 2.4.1, la formule précédente est soit la formule $\exists \varepsilon vrai$, soit la formule $\exists \varepsilon faux$. En suivant les mêmes étapes et en utilisant la propriété 3.1.5, nous montrons le reste de cette propriété pour les formules de base de la forme $\forall \bar{x}\, \overline{\alpha}$.

3.3 La procédure de décision

Soit φ une proposition. Le calcul de la valeur de vérité de φ dans T s'effectue de la manière suivante :

(1) Transformer φ en une formule positive ϕ.

(2) Appliquer les règles de réécriture de la figure 1 sur les sous-formules positives de ϕ en considérant que les connecteurs $\wedge$ et $\vee$ sont associatifs, commutatifs et idempotents [2].

(3) Répéter la deuxième étape jusqu'à atteindre un point fixe (aucune règles ne peut être appliquée). Nous montrons par la propriété 3.2.2 que nous obtenons alors soit la formule *vrai*, soit la formule *faux*.

Comment ça marche ? notre algorithme utilise la décomposition pour éliminer des quantificateurs quand cela est possible et retravaille la formule dans le cas ou il n'y pas d'élimination possible de quantificateurs dans une sous formule quantifiée. Tout cela a pour objectif de générer dès que c'est possible des formules de bases sous forme de propositions que l'on va réduire directement à vrai ou à faux. Plus précisément, partant d'une formule φ sans variables libres, les règles (1),...(4) prépare le phénomène de décomposabilité en transformant φ en une formule contenant des sous formules quantifiée de la forme $\exists \overline{x}\, \alpha \wedge \bigwedge_{i\in I} \varphi_i$ ou $\forall \overline{x}\, \overline{\alpha} \vee \bigvee_{i\in I} \varphi_i$ où φ_i est une formule positive. Les règles (7) et (8) vérifient que le premier niveau de la quantification n'est pas équivalent à vrai ou à faux. Les règles (9) et (10) décomposent le premier niveau de la quantification et propage la troisième partie de la décomposition (les formules qui appartiennent à A'''). Toutes ces étapes sont répétées jusqu'à ce que l'on ne puisse plus propager des formule de A'''. Les règles (11), (12) suivies par les règles (13) et (14) (tous avec $I = \emptyset$) éliminent les formules qui appartiennent à A''' puis ceux qui appartiennent à A'' à partir des formules les plus imbriquées. Une fois cette élimination effectuée, seules les formules de A' apparaîtrons dans le dernier niveau imbriqué de nos formules. Les règles (11),...,(14) peuvent maintenant être appliquées avec $I \neq \emptyset$ et créent des formules de la forme $\exists \overline{x}'\, \alpha' \wedge \bigwedge_{i\in I'} \overline{\beta'_i}$ ou $\forall \overline{x}'\, \overline{\alpha}' \vee \bigvee_{i\in I'} \beta'_i$. Les règles (5) et (6) ainsi que les autres règles peuvent être maintenant re-appliquées. après application fini de nos règles, nous obtenons une combinaison booléenne de formules de la forme $\exists \overline{x}'\, \alpha' \wedge \bigwedge_{i\in I'} \overline{\beta'_i}$ ou $\forall \overline{x}'\, \overline{\alpha}' \vee \bigvee_{i\in I'} \beta'_i$. Du fait que ces formules ne contiennent pas de variables libres alors, par la propriété 3.2.2, chaque niveau est de l'une des formes suivantes :

$$(\exists \varepsilon\ vrai),\ (\exists \varepsilon\ faux),\ (\forall \varepsilon\ vrai), (\forall \varepsilon\ faux).$$

Par conséquent, après application finie des règles (15),...,(18) nous obtenons soit la formule vrai, soit la formule faux.

Correction des règles

Montrons que pour chaque règle de la forme $p \Longrightarrow p'$ on a $T \models p \leftrightarrow p'$. Les règles (1),...,(8), (15),...,(18) sont évidente et se déduisent à partir des propriétés de base de la logique du premier ordre. Les autres règles sont de nouvelles propriétés des théories décomposables et méritent une preuve formelle.

3.3.1 Preuve de la règle (9)

(9) $\exists \overline{x}\, \alpha \wedge \bigwedge_{i\in I} \varphi_i \quad \Rightarrow \quad (\exists \overline{x}'\, \alpha') \wedge (\forall \overline{x}'\, \overline{\alpha'} \vee \exists \overline{x}''\, \alpha'' \wedge \bigwedge_{i\in I} (\forall \overline{x}'''\, \overline{\alpha'''} \vee \varphi_i))$

D'après les condition d'application de cette règle, la formule $\exists \bar{x}\, \alpha$ est équivalente dans T à une formule décomposée de la forme $\exists \bar{x}'\, \alpha' \wedge (\exists \bar{x}''\, \alpha'' \wedge (\exists \bar{x}'''\, \alpha'''))$. Donc, la formule à gauche de cette règle est équivalente T à la formule

$$\exists \bar{x}'\, \alpha' \wedge (\exists \bar{x}'' \alpha'' \wedge (\exists \bar{x}''' \alpha''' \wedge \bigwedge_{i\in I} \varphi_i)).$$

Du fait que $\exists \bar{x}'''\, \alpha''' \in A'''$, alors d'après le point 4 de la définition 2.4.1 nous avons $T \models \exists! \bar{x}''' \alpha'''$, donc en utilisant la propriété 2.2.4, la formule précédente est équivalente dans T à

$$\exists \bar{x}'\, \alpha' \wedge (\exists \bar{x}'' \alpha'' \wedge \bigwedge_{i\in I} \neg(\exists \bar{x}''' \alpha''' \wedge \neg\varphi_i)).$$

D'après le deuxième point de la définition 2.4.1 nous avons $T \models \exists ? \bar{x}' \alpha'$, donc en utilisant la propriété 2.2.3, la formule précécente est équivalent dans T à

$$(\exists \bar{x}'\, \alpha') \wedge \neg(\exists \bar{x}'\, \alpha' \wedge \neg(\exists \bar{x}'' \alpha'' \wedge \bigwedge_{i\in I} \neg(\exists \bar{x}''' \alpha''' \wedge \neg\varphi_i))),$$

c'est-à-dire à

$$(\exists \bar{x}'\, \alpha') \wedge (\forall \bar{x}'\, (\neg\alpha') \vee (\exists \bar{x}'' \alpha'' \wedge \bigwedge_{i\in I} (\forall \bar{x}'''(\neg\alpha''') \vee \varphi_i))),$$

qui d'après la propriété 3.1.5 est équivalente dans T à

$$(\exists \bar{x}'\, \alpha') \wedge (\forall \bar{x}'\, \overline{\alpha}' \vee \exists \bar{x}'' \alpha'' \wedge \bigwedge_{i\in I} (\forall \bar{x}''' \overline{\alpha}''' \vee \varphi_i)).$$

2. $p \vee p \leftrightarrow p$ et $p \wedge p \leftrightarrow p$.

Distribution

$$\begin{array}{lrcl}
(1) & \exists\overline{x}\,\varphi_1 \wedge (\varphi_2 \vee \varphi_3) & \Rightarrow & \exists\overline{x}\,(\varphi_1 \wedge \varphi_2) \vee (\varphi_1 \wedge \varphi_3) \\
(2) & \forall\overline{x}\,\varphi_1 \vee (\varphi_2 \wedge \varphi_3) & \Rightarrow & \forall\overline{x}\,(\varphi_1 \vee \varphi_2) \wedge (\varphi_1 \vee \varphi_3) \\
(3) & \exists\overline{x}\,\varphi_1 \vee \varphi_2 & \Rightarrow & \exists\overline{x}\,\varphi_1 \vee \exists\overline{x}\,\varphi_2 \\
(4) & \forall\overline{x}\,\varphi_1 \wedge \varphi_2 & \Rightarrow & \forall\overline{x}\,\varphi_1 \wedge \forall\overline{x}\,\varphi_2
\end{array}$$

Remonté des quantificateurs pour préparation à la décomposition

$$\begin{array}{lrcl}
(5) & \exists\overline{x}'\,\alpha' \wedge (\exists\overline{y}'\beta' \wedge \bigwedge_{i\in I}(\forall\overline{z}'\,\overline{\lambda_i'})) \wedge \varphi_1 & \Rightarrow & \exists\overline{x}'\overline{y}'\,\alpha' \wedge \beta' \wedge \bigwedge_{i\in I}(\forall\overline{z}'\,\overline{\lambda_i'}) \wedge \varphi_1 \\
(6) & \forall\overline{x}'\,\overline{\alpha}' \vee (\forall\overline{y}'\overline{\beta}' \vee \bigvee_{i\in I}(\exists\overline{z}'\,\lambda_i')) \vee \varphi_1 & \Rightarrow & \forall\overline{x}'\overline{y}'\,\overline{\alpha}' \vee \overline{\beta}' \vee \bigvee_{i\in I}(\exists\overline{z}'\,\lambda_i') \vee \varphi_1
\end{array}$$

Résolution locale par décomposition en échec

$$\begin{array}{lrcl}
(7) & \exists\overline{x}\,\alpha \wedge \bigwedge_{i\in I}\varphi_i & \Rightarrow & \mathit{faux} \\
(8) & \forall\overline{x}\,\overline{\alpha} \vee \bigvee_{i\in I}\varphi_i & \Rightarrow & \mathit{vrai}
\end{array}$$

Décomposition sans échec

$$\begin{array}{lrcl}
(9) & \exists\overline{x}\,\alpha \wedge \bigwedge_{i\in I}\varphi_i & \Rightarrow & (\exists\overline{x}'\,\alpha') \wedge (\forall\overline{x}'\,\overline{\alpha'} \vee \exists\overline{x}''\,\alpha'' \wedge \bigwedge_{i\in I}(\forall\overline{x}'''\,\overline{\alpha'''} \vee \varphi_i)) \\
(10) & \forall\overline{x}\,\overline{\alpha} \vee \bigvee_{i\in I}\phi_i & \Rightarrow & (\forall\overline{x}'\,\overline{\alpha'}) \vee (\exists\overline{x}'\,\alpha' \wedge \forall\overline{x}''\,\overline{\alpha}'' \vee \bigvee_{i\in I}(\exists\overline{x}'''\,\alpha''' \wedge \phi_i))
\end{array}$$

Propagation des formules quantifiées de A''' + élimination (lorsque $I = \emptyset$)

$$\begin{array}{lrcl}
(11) & \exists\overline{x}'''\,\alpha''' \wedge \bigwedge_{i\in I}\overline{\beta_i} & \Rightarrow & \bigwedge_{i\in I}\forall\overline{x}'''\overline{\alpha'''} \vee \overline{\beta}_i \\
(12) & \forall\overline{x}'''\,\overline{\alpha}''' \vee \bigvee_{i\in I}\beta_i & \Rightarrow & \bigvee_{i\in I}\exists\overline{x}'''\alpha''' \wedge \beta_i
\end{array}$$

Simplification en A' + élimination des formules quantifiées de A''

$$\begin{array}{lrcl}
(13) & \exists\overline{x}\,\alpha \wedge \bigwedge_{i\in I}\overline{\beta_i} & \Rightarrow & \exists\overline{x}'\,\alpha' \wedge \bigwedge_{i\in I'}\overline{\beta_i} \\
(14) & \forall\overline{x}\,\overline{\alpha} \vee \bigvee_{i\in I}\beta_i & \Rightarrow & \forall\overline{x}'\,\overline{\alpha}' \vee \bigvee_{i\in I'}\beta_i
\end{array}$$

Propagation du *faux* et du *vrai*

$$\begin{array}{lrcl}
(15) & \exists\overline{x}\,\varphi \wedge \mathit{faux} & \Rightarrow & \mathit{faux} \\
(16) & \forall\overline{x}\,\varphi \wedge \mathit{faux} & \Rightarrow & \mathit{faux} \\
(17) & \exists\overline{x}\,\varphi \vee \mathit{vrai} & \Rightarrow & \mathit{vrai} \\
(18) & \forall\overline{x}\,\varphi \vee \mathit{vrai} & \Rightarrow & \mathit{vrai}
\end{array}$$

Dans toute ces règles, I est un ensemble fini éventuellement vide [3], les formules φ_i et ϕ_i sont des formules positives et $\alpha \in AT$.

- Dans les règles (1),...,(4), le vecteur $\bar{x}$ n'est pas vide.
- Dans les règles (5) et (6), les formules $(\exists x'\,\alpha')$, $(\exists\overline{y}'\beta')$ ainsi que chaque $(\exists\bar{z}_i\,\lambda_i)$ appartiennent à A'.
- Dans les règles (7) et (8), la formule de base $\exists\bar{x}\,\alpha$ est équivalente à une formule décomposée de la forme $(\exists\bar{x}'\,\mathit{faux} \wedge (\exists\bar{x}''\,\alpha'' \wedge (\exists\bar{x}'''\,\alpha''')))$.
- Dans les règles (9) et (10), pour tout $i \in I$, $\varphi_i \notin AT$ et $\overline{\phi_i} \notin AT$. De plus, la formule de base $\exists\bar{x}\,\alpha$ est équivalente à une formule décomposée de la forme $(\exists\bar{x}'\,\alpha' \wedge (\exists\bar{x}''\,\alpha'' \wedge (\exists\bar{x}'''\,\alpha''')))$, avec :
  - $\alpha' \neq \mathit{faux}$,
  - $\exists\bar{x}'''\,\alpha''' \neq \exists\varepsilon\,\mathit{vrai}$.
- Dans les règles (11) et (12) :
  - Pour tout $i \in I$, on a $\beta_i \in A'$.
  - $(\exists\bar{x}'''\,\alpha''') \in A'''$.
- Dans les règles (13) et (14) :
  - La formule $\exists\bar{x}\,\alpha$ n'est pas un élément de A' et est équivalente dans T à une formule décomposée de la forme $\exists\bar{x}'\,\alpha' \wedge (\exists\bar{x}''\,\alpha'' \wedge (\exists\varepsilon\,\mathit{vrai}))$ avec $\alpha' \neq \mathit{faux}$.
  - Pour tout $i \in I$, on a $\beta_i \in A'$.
  - I' est l'ensemble des $i \in I$ telles que β_i ne contienne pas d'occurrences libres d'aucune des variables du vecteur $\bar{x}''$.

Fig 1. Transformation d'une formule positive.

3.3.2 Preuve de la règle (10)

$$(10)\ \ \forall \bar{x}\,\overline{\alpha} \vee \bigvee_{i\in I}\phi_i \quad \Rightarrow \quad (\forall \bar{x}'\,\overline{\alpha'}) \vee (\exists \bar{x}'\,\alpha' \wedge \forall \bar{x}''\,\overline{\alpha''} \vee \bigvee_{i\in I}(\exists \bar{x}'''\,\alpha''' \wedge \phi_i))\ .$$

Dans la preuve précédente, nous avons montré que le membre gauche de la règle (9) est équivalent aux membre droit. Donc en mettant une négation sur les deux extrémités de l'équivalence, nous obtenons :

$$T \models\ \neg(\exists \bar{x}\,\alpha \wedge \bigwedge_{i\in I}\varphi_i) \quad \leftrightarrow \quad \neg((\exists \bar{x}'\,\alpha') \wedge (\forall \bar{x}'\,\overline{\alpha'} \vee \exists \bar{x}''\,\alpha'' \wedge \bigwedge_{i\in I}(\forall \bar{x}'''\,\overline{\alpha'''} \vee \varphi_i))).$$

En descendant la négation et en utilisant la propriété 3.1.5 nous obtenons

$$T \models\ \forall \bar{x}\,\overline{\alpha} \vee \bigvee_{i\in I}\phi_i \quad \leftrightarrow \quad (\forall \bar{x}'\,\overline{\alpha'}) \vee (\exists \bar{x}'\,\alpha' \wedge \forall \bar{x}''\,\overline{\alpha''} \vee \bigvee_{i\in I}(\exists \bar{x}'''\,\alpha''' \wedge \phi_i)),$$

où ϕ_i est la formule $\neg\varphi_i$. D'après les conditions d'application de la règle (9) nous avons $\varphi_i \notin AT$, par conséquent, d'après la propriété 3.1.5 nous obtenons $\overline{\phi_i} \notin AT$.

3.3.3 Preuve de la règle (11)

$$(11)\ \ \exists \bar{x}'''\,\alpha''' \wedge \bigwedge_{i\in I}\overline{\beta_i} \quad \Rightarrow \quad \bigwedge_{i\in I}\forall \bar{x}'''\overline{\alpha'''} \vee \overline{\beta}_i.$$

D'après la propriété 3.1.5, le membre gauche de cette est équivalent dans T à

$$(\exists \bar{x}'''\,\alpha''' \wedge \bigwedge_{i\in I}\neg\beta_i).$$

Du fait que $\exists \bar{x}'''\,\alpha''' \in A'''$, alors d'après le quatrième point de la définition 2.4.1, on a $T \models \exists! \bar{x}'''\alpha'''$. Donc, d'après le propriété 2.2.4, la formule précédente est équivalente dans T à

$$\bigwedge_{i\in I}\neg(\exists \bar{x}'''\,\alpha''' \wedge \beta_i),$$

c'est-à-dire à

$$\bigwedge_{i\in I}(\forall \bar{x}'''\,\neg\alpha''' \vee \neg\beta_i),$$

qui d'après la propriété 3.1.5 est équivalente à

$$\bigwedge_{i\in I}(\forall \bar{x}'''\,\overline{\alpha'''} \vee \overline{\beta_i}).$$

3.3.4 Preuve de la règle (12)

$$(12)\ \ \forall \bar{x}'''\,\overline{\alpha}''' \vee \bigvee_{i\in I}\beta_i \Rightarrow \bigvee_{i\in I}\exists \bar{x}'''\alpha''' \wedge \beta_i.$$

Dans la preuve précédente, nous avons montré que le membre gauche de la règle (9) est équivalent aux membre droit. Donc en mettant une négation sur les deux extrémités de l'équivalence, nous obtenons :

$$T \models \neg(\exists \bar{x}'''\,\alpha''' \wedge \bigwedge_{i\in I}\overline{\beta_i}) \leftrightarrow \neg(\bigwedge_{i\in I}\forall \bar{x}'''\overline{\alpha'''} \vee \overline{\beta}_i).$$

En descendant la négation et en utilisant la propriété 3.1.5 nous obtenons

$$T \models \forall \bar{x}'''\,\overline{\alpha'''} \vee \bigvee_{i\in I}\beta_i \leftrightarrow \bigvee_{i\in I}\exists \bar{x}'''\alpha''' \wedge \beta_i.$$

3.3.5 Preuve de la règle (13)

$$(13)\ \ \exists \bar{x}\,\alpha \wedge \bigwedge_{i\in I}\overline{\beta_i} \quad \Rightarrow \quad \exists \bar{x}'\,\alpha' \wedge \bigwedge_{i\in I'}\overline{\beta_i}$$

avec :

- la formule $\exists \bar{x}\,\alpha$ n'est pas un élément de A' et est équivalente dans T à une formule décomposée de la forme $\exists \bar{x}'\,\alpha' \wedge (\exists \bar{x}''\,\alpha'' \wedge (\exists \varepsilon\ vrai))$ avec $\alpha' \neq faux$.
- Pour tout $i \in I$, nous avons $\beta_i \in A'$.
- I' est l'ensemble des $i \in I$ tels que β_i ne contienne pas d'occurrences libres d'aucune des variables du vecteur $\bar{x}''$.

D'après les conditions précédentes, le membre gauche de la règle (13) est équivalent dans T à

$$\exists \bar{x}'\,\alpha' \wedge (\exists \bar{x}''\,\alpha'' \wedge \bigwedge_{i\in I}\overline{\beta_i}).$$

Notons I_1, l'ensemble des $i \in I$ tel que x''_n n'ait pas d'occurrences libres dans β_i. La formule précédente est donc équivalente dans T à

$$\exists \bar{x}'\,\alpha' \wedge (\exists x''_1 ... \exists x''_{n-1}\begin{bmatrix}(\bigwedge_{i\in I_1}\overline{\beta_i}) \wedge \\ (\exists x''_n\,\alpha'' \wedge \bigwedge_{i\in I-I_1}\overline{\beta_i})\end{bmatrix}). \quad (8)$$

Du fait que $\exists \bar{x}''\alpha'' \in A''$ et $\beta_i \in A'$ pour tout $i \in I - I_1$, alors d'après la propriété 2.3.2 et les conditions 2 et 3 de la définition 2.4.1, la formule (8) est équivalente dans T à

$$\exists \bar{x}'\,\alpha' \wedge (\exists x''_1 ... \exists x''_{n-1}\,((\bigwedge_{i\in I_1}\overline{\beta_i}) \wedge vrai)). \quad (9)$$

En répétant les trois étapes précédentes $(n-1)$ fois, en notons I_k l'ensemble des $i \in I_{k-1}$ tels que $x''_{(n-k+1)}$ ne contienne pas d'occurrences libres dans β_i, et en utilisant $(n-1)$ fois la propriété 2.3.3, la formule précédente est équivalente dans T à

$$\exists \bar{x}'\,\alpha' \wedge \bigwedge_{i\in I_n}\overline{\beta_i}.$$

3.3.6 Preuve de la règle (14)

$$(14)\ \ \forall \overline{x}\,\overline{\alpha} \vee \bigvee_{i\in I} \beta_i \Rightarrow \forall \overline{x}'\,\overline{\alpha}' \vee \bigvee_{i\in I'} \beta_i.$$

Dans la preuve précédente, nous avons montré que le membre gauche de la règle (9) est équivalent aux membre droit. Donc en mettant une négation sur les deux extrémités de l'équivalence, nous obtenons :

$$T \models \neg(\exists \overline{x}\,\alpha \wedge \textstyle\bigwedge_{i\in I} \overline{\beta_i}) \quad \Rightarrow \quad \neg(\exists \overline{x}'\,\alpha' \wedge \textstyle\bigwedge_{i\in I'} \overline{\beta_i}).$$

En descendant la négation et en utilisant la propriété 3.1.5 nous obtenons

$$T \models \forall \overline{x}\,\overline{\alpha} \vee \textstyle\bigvee_{i\in I} \beta_i \quad \leftrightarrow \quad \forall \overline{x}'\,\alpha' \vee \bigvee_{i\in I'} \beta_i.$$

Exemple 3.3.7 *Calculons la valeur de vérité de la formule φ_1 dans la théorie des arbres finis ou infinis [7] :*

$$\exists x \forall y\, \exists z\, z = f(y) \wedge x = f(x) \vee (x = f(y) \wedge z = f(z))$$

D'après la règle (3), la formule précédente est équivalente à

$$\exists x \forall y\, (\exists z\, z = f(y) \wedge x = f(x)) \vee (\exists z\, x = f(y) \wedge z = f(z)).$$

En appliquant la règle (9) avec $I = \emptyset$ sur $(\exists z\, z = f(y) \wedge x = f(x))$ et aussi sur $(x = f(y) \wedge z = f(z))$, nous obtenons la formule équivalente suivante

$$\exists x \forall y\, ((x = f(x)) \wedge (x = f(x))) \vee ((x = f(y)) \wedge ((x = f(y))).$$

En effet :

- *la formule $(\exists z\, z = f(y) \wedge x = f(x))$ est équivalente à une formule décomposée de la forme $(\exists \varepsilon\, x = f(x) \wedge (\exists \varepsilon \text{ vrai} \wedge (\exists z\, z = f(y)))$.*
- *la formule $(\exists z\, x = f(y) \wedge z = f(z))$ est équivalente à une formule décomposée de la forme $(\exists \varepsilon\, x = f(y) \wedge (\exists \varepsilon \text{ vrai} \wedge (\exists z\, z = f(z)))$.*

Du fait que nos règles considère les connecteurs $\wedge$ et $\vee$ comme étant associatifs, commutatifs et idempotents alors la formule précédente est équivalente à

$$\exists x \forall y\, x = f(x) \vee x = f(y).$$

Du fait que $\exists x \text{ vrai} \in A''$, alors la règle (13) peut être appliquées. La formule précédente est donc équivalente à la conjonction vide, c'est-à-dire, à la formule vrai. La procédure classique des théories décomposables [8] aurait consommé beaucoup plus de temps et d'espace avant d'arriver au même résultat.

4 Discussions

Qu'avons nous fait ? Nous avons présenté dans ce papier une procédure de décision pour les théories décomposables fonctionnelles. La procédure de décision classique avait recours à une normalisation de formules qui structurait la formule sous forme d'arbres dont la profondeur diminuait au prix d'une règle exponentielle en temps et espace. Nous avons alors essayé d'éviter cette normalisation en utilisant la notion de contraintes duales ainsi que de nouvelles propriétés des théories fonctionnelles décomposables.

En pratique ? Nous avons réalisé des séries de benchmarks sur deux théories différentes à base d'arbres et de rationnels munis d'addition et de soustraction. Nous nous sommes alors rendu compte que pour des formules aléatoires ayant des alternations de quantificateurs sur des combinaisons booléennes (similaire par exemple à celle de l'exemple 3.3.7), le temps d'exécution ainsi que l'espace mémoire était considérablement réduit par rapport à la procédure de décision classique. Nous avons également détecté le phénomène suivant : si la formule de départ ne contient pas de contradiction dans ses sous formules alors notre algorithme est beaucoup plus rapide que celui qui utilise les formules normalisées [6]. L'explication est simple : l'algorithme [6] procède par développement en profondeur, c'est-à-dire qu'il part d'une formule du premier ordre, la transforme en formule normalisée sous forme d'arbre avec une profondeur d et commence par diminuer la profondeur de l'arbre en utilisant une règle très coûteuse en temps et espace. Cette dernière est appliquée une fois un test de propagation effectué ; ce qui permet d'éliminer les sous formules normalisées qui contredisent d'autre sous formules comme par exemple :

$$\neg(\exists \varepsilon x = 0 \wedge \neg(\exists \varepsilon y = 0 \wedge \neg(\exists \varepsilon x = 1))).$$

Dans cette formule, la propagation de la contrainte $x = 0$ jusqu'à la feuille de l'arbre normalisé permettra d'éliminer le dernier niveau de la formule est affichera directement le résultat $x = 0 \wedge y \neq 0$ sans appliquer la règle de diminution de profondeur. Or, si le résultat de la propagation est nul alors l'algorithme part dans une série de distributions très coûteuses due à la normalisation. Cette normalisation est elle même responsable de la génération d'une formule beaucoup plus grande et complexe que la formule initiale avant même de commencer la résolution proprement dite. Notre nouvelle procédure de décision ne normalise pas la formule initiale et effectue un traitement en largeur et non pas en profondeur en utilisant les règles de la décomposabilité, ce qui s'est révélé plus efficace dans plus de 70% des séries de benchmarks que nous avons réalisées.

Faut il alors supprimer complètement la normalisation ? La réponse à cette question est non. Il ne faut pas oublier que 'calculer la valeur de vérité des propositions c'est bien ; mais réaliser un solveur de contraintes avec variables libres c'est mieux ! " En effet, on modélise souvent des problèmes avec des formules du premier ordre qui contiennent des variables libres et on cherche à savoir les valeurs des variables libres qui vont satisfaire cette formule dans un modèle particulier : arbres, entiers, intervalles,...etc. Pour cela, notre procédure de décision va créer une combinaison booléenne de formules de base qui ne pourra être simplifiée en vrai ou en faux du fait de l'existence de variables libres. On aura alors à la fin une combinaison complexe équivalente à la formule de départ mais dans laquelle les solutions des variables libres sont loin d'être évidentes à extraire. La normalisation apporte justement une solution à ce problème et permet de maintenir une certaine forme restreinte de formules qui permet de dégager simplement l'ensembles des solutions des variables libres. Ce qui explique pourquoi nous avons gardé cette normalisation pour la version révisée [6] de la procédure de décision classique des théories décomposables [8].

Perspectives Ce papier ainsi présenté participe à la compréhension des mécanismes de base pour la résolution de contraintes du premier ordre, mais un long chemin de recherche reste encore à parcourir si l'on veut aboutir à de puissants solveurs sur des cas pratiques complexes. Nous sommes actuellement en train de développer une page web sur la décomposabilité. En plus d'une large bibliothèque de théories décomposables, on y trouvera une application qui offre à l'utilisateur la possibilité de saisir le code C++ ou Java permettant de décomposer toute formule de la forme $\exists \bar{x}\, \alpha$ et on lui générera alors automatiquement le solveur de contraintes du premier ordre [6] ainsi que la procédure de décision présentée dans ce papier. Nous sommes également entrain de travailler sur des cas plus concrets de benchmarks. En effet, jusqu'à la nous avons testé nos solveurs uniquement sur des exemples générés aléatoirement. Deux théories en particulier attirent notre attention : Presburger's [9] et les tableaux [11].

D'autre part, nous nous sommes restreint dans ce papier aux théories fonctionnelles. On peut dépasser cette restriction et la rendre plus générale en considérant les théories *réversibles*. Ce sont des théories que nous avons développé et dans lesquelles nous pouvons éviter la négation sur les formules atomiques en générant une formule équivalente qui ne contient pas d'occurrence de négation. Le prix à payer est souvent la création d'une combinaison booléenne de formules atomiques. Nous examinons actuellement les limites de cette approche et essayons de trouver des bornes de complexité.

Références

[1] Bradely A-R., Manna, Z. The Calculus of Computation : Decision Procedures with Applications to Verification. Livre. Springer Verlag, ISBN : 978-3-540-74112-1. 2007.

[2] Benedetti M., Lallouet A., Vautard J. Quantified. Constraint Optimization. Dans les actes de CP 2008. LNCS, vol 5202, pp. 463-477. 2008.

[3] Benedetti M., Lallouet A., Vautard J. Quantified. QCSP Made Practical by Virtue of Restricted Quantification. Dans les actes de IJCAI 2007, pp.38-43. 2007.

[4] Bordeaux L. Résolution de problèmes combinatoires modélisés par des contraintes quantifiées. Thèse de doctorat de l'université de Nantes. 2003.

[5] Dao T-B-H. Résolution de contraintes du premier ordre dans la théorie des arbres finis ou infinis, T-B-H. Dao, Thèse de Doctorat de l'université de la Méditerranée. 2000.

[6] Djelloul K. A full first-order constraints solver for decomposable theories. Annals of Mathematics and Artificial Intelligence. A paraitre. 2009.

[7] Djelloul K., Dao T., Fruehwirth T. Theory of finite or infinite trees revisited. Theory and practice of logic programming (TPLP). Vol 8(4) : 431-489. 2008.

[8] Djelloul K. Decomposable theories. Theory and practice of logic programming (TPLP). Vol 7(5) : 583-632. 2007.

[9] Oppen, D. A $2^{2^{2^n}}$ Upper Bound on the Complexity of Presburger Arithmetic. Journal of Comput. Syst. Sci. Vol 16(3) : 323-332. 1978.

[10] Rossi F., Beek P., Walsh T. Handbook of Constraint Programming. Livre. Elsevier, ISBN : 978-0-444-52726-4. 2006.

[11] Ghilardi S., Nicolini E., Ranise S., Zucchelli D. Towards SMT Model Checking of Array-Based Systems. Dans les actes de IJCAR 2008. LNCS vol 5195, pp. 67-82. 2008.

[12] SMT web-page. Page web de la communauté SMT. http ://combination.cs.uiowa.edu/smtlib/

[13] Verger G., Bessiere C. Guiding Search in QCSP+ with Back-Propagation. Dans les actes de CP 2008. LNCS, vol 5202, pp. 175-189. 2008

Problèmes d'optimisation avec des contraintes quantifiées

Marco Benedetti, Arnaud Lallouet and Jérémie Vautard
Université d'Orléans – LIFO
BP 6759, F-45067 Orléans
{prenom.nom}@univ-orleans.fr

Résumé

Une solution d'un problème de contraintes quantifiées (QCSP) peut être considérée comme une stratégie, qui est une représentation de la manière dont le joueur existentiel réagit au coups du joueur universel. Cependant, ces stratégies ne sont pas toutes équivalentes et certaines peuvent être préférées aux autres. Dans ce papier, nous définissons des problèmes d'optimisations de contraintes quantifiées (QCOP - Quantified constraint Optimization Problem) dans lesquels l'ordre de préférence des sous-stratégies peut être défini sur plusieurs niveaux. Nous montrons que ce formalisme permet de représenter des problèmes de décision hiérarchiques comme les Jeux de Stackelberg ou les problèmes de programmation multi-niveaux.

1 Introduction

La programmation par contraintes quantifiées permet d'exprimer de façon naturelle et concise des problèmes hors de portée des CSP, tels que les jeux à deux joueurs [1, 2, 3] et autres problèmes avec adversaire [4], la vérification de modèles, ou l'ordonnancement robuste par rapport à l'environnement [5].

Résoudre un CSP consiste à trouver une valeur pour chacune de ses variables de manière à ce que toutes les contraintes soient satisfaites. Dans un CSP quantifié (QCSP), une variable peut être quantifiée de manière universelle sur son domaine, ce qui permet d'exprimer une incertitude sur la valeur que peut prendre cette variable, ce qui permet de modéliser le comportement possible d'un adversaire, ou une incertitude sur l'environnement. Un QCSP est vrai si on peut trouver des valeurs pour les variables existentielles restantes qui soient consistantes avec toutes les valeurs des universelles. Ainsi, la notion de solution d'un QCSP est une famille de fonctions de Skolem appelée *stratégie* qui associe une valeur à chaque variable existentielle en fonction des variables universelles précédentes. Une telle stratégie n'est pas un objet compact : sa taille est dans le cas général exponentielle en le nombre de variables. Un QCSP est vrai s'il possède au moins une stratégie *gagnante*, c'est à dire dans laquelle les valeurs données aux variables existentielles pour chaque affectation possible des variables universelles satisfont toutes les contraintes.

Toutes les stratégies gagnantes sont-elles équivalentes ? Dans la modélisation d'un jeu, il pourrait être intéressant de trouver la stratégie permettant de gagner le plus rapidement possible. Dans beaucoup de cas. les stratégies peuvent être évaluées et ordonnées selon une préférence donnée. Le moyen le plus facile d'exprimer une préférence sur les solutions est de définir une fonction de l'ensemble des stratégies vers un ensemble ordonné et de choisir une stratégie qui optimise cette valeur. C'est précisément ce que nous faisons ici. Dans les CSP, on peut exprimer une telle optimisation en demandant une solution qui maximise ou minimise telle variable du problème. Cependant les stratégies sont des objets complexes, et nous proposons, pour exprimer une telle fonction d'optimisation, de définir un langage appelé QCOP$^+$, basé que les QCSP$^+$ [3]. Ce langage suit la structure récursive de la formule d'un QCSP

$$\begin{array}{l}\exists X \in D_X \,.\, [C_1(X)] \\ \quad \forall Y \in D_Y \,.\, [C_2(X,Y)] \\ \quad\quad \exists Z \in D_Z \,.\, [C_3(X,Y,Z)] \\ \quad\quad\quad C(X,Y,Z) \\ \quad\quad \min(Z) \\ \quad s : sum(Z) \\ \max(s)\end{array}$$

FIG. 1 – Exemple de QCOP$^+$

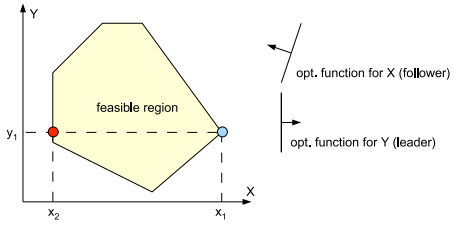


FIG. 2 – Optimum pour le meneur seul, et réponse du suiveur

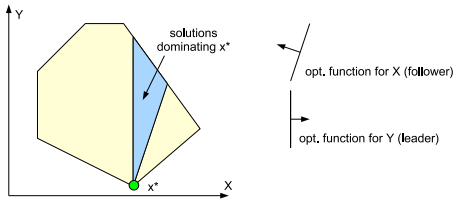


FIG. 3 – Équilibre optimal

et permet d'optimiser sur les variables existentielles et de calculer des agrégats au niveau des variables universelles. La figure 1 donne un exemple d'un problème d'optimisations entrelacées en QCOP$^+$ (chaque optimisation et calcul d'agrégat est associé au quantificateur ayant la même indentation). Le but est de trouver une valeur X qui maximise la valeur de l'agrégat s calculé à partir de la sous-stratégie. Au niveau inférieur suivant, X est instanciée à la valeur choisie et un aggrégat est calculé en sommant les valeurs de Z retournées pour chaque valeur de Y. A un niveau encore inférieur, la valeur de Z retournée est la plus petite valeur possible telle que $C(X, Y, Z)$ est vraie. Les conditions C_1 à C_3 viennent restreindre les valeurs possible d'une variable de manière dynamique. Ainsi, la somme calculée dans s est la somme des valeurs minimales de Z. Cet agrégat est alors évalué pour toutes les valeurs de X, et on retourne une stratégie affectant une valeur à X qui maximise s. Le nombre de niveaux possibles n'est pas limité.

En regardant dans la littérature sur les QCSP, nous n'avons pas trouvé de notion générale d'optimisation. Cependant, des problèmes de ce genre sont étudiés depuis les années 1970 en programmation mathématique sous le nom de problèmes de programmation *bi-niveaux* ou *multi niveaux*. [6]. Les programmes bi-niveaux sont utilisés pour résoudre des problèmes de décision sous la forme de jeux de Stackelberg, qui est un modèle d'oligopole en théorie des jeux [7]. Dans ces problèmes sont représentés deux acteurs agissant de manière séquentielle, mais n'ayant aucun pouvoir l'un sur l'autre. Le premier à décider est appelé le meneur, et le second (appelé le suiveur) prend acte de cette décision et adapte sa propre décision en fonction. Le problème est que le meneur et le suiveur ont des fonctions d'objectif différentes, qui ne sont pas forcément concordantes ni opposées. Par exemple, le meneur peut être une agence gouvernementale qui partage des fonds disponibles entre plusieurs entités dont chacune a la liberté d'utiliser sa dotation comme elle l'entend. L'exemple suivant, pris dans [8], montre que des objectifs en conflit peuvent amener à un équilibre non-optimal. Dans la situation exposée en figure 2, le choix du couple (x_1, y_1) qui serait optimal sans le suiveur devient considérablement sous-optimal quand ce dernier entre en jeu, menant à la solution (x_2, y_1). L'équilibre x^* est montré en figure 3, où l'on peut noter qu'il existe des solution strictement dominantes et pour le meneur, et pour le suiveur, qui ne peuvent jamais être atteintes sans consensus. Nous nous référons à [9] pour une étude extensive sur la programmation bi-niveau. Le terme "programmation multi-niveaux" s'applique dès lors qu'il y a plus de deux niveaux de décisions hiérarchiques.

Nous avons intégré le formalisme des QCOP$^+$ dans le solveur QeCode [10] basé sur Gecode [11], ainsi qu'une extraction simple de la stratégie gagnante, ce qui est aussi utile pour les QCSP$^+$ classiques : dans beaucoup de situations, une simple réponse "booléenne" ne suffit pas étant donné que l'utilisateur désire obtenir les valeurs des variables (comme pour les CSP). L'extraction des stratégies a été présentée dans [12] dans le cadre des QBF. Ce papier présente les QCSP, les QCSP$^+$, et les QCOP$^+$ ainsi que leur résolution. Nous y présentons l'algorithme de recherche et en étudions des premières variantes de branch-and-bound. Enfin, nous donnons quelques exemples de modélisation de problèmes bi-niveaux.

2 QCSP

Notations. soit V un ensemble de variables et $D = (D_X)_{X \in V}$ la famille de leurs domaines. On rappelle qu'une famille est une fonction d'un ensemble indicé dans un ensemble. Pour un sous-ensemble $W \subseteq V$, on note D^W l'ensemble des n-uplets de W, c'est à dire $\Pi_{X \in W} D_X$. La projection d'un n-uplet (ou d'un ensemble de n-uplets) sur une variable (ou un ensemble de variables) est notée $|$. Par exemple, pour $t \in D^V$, $t|_W = (t_x)_{X \in W}$ et pour $E \subseteq D^V$, $E|_W = \{t|_W \mid t \in E\}$. Pour $W, U \subseteq V$, la jointure de $A \subseteq D^W$ et $B \subseteq D^U$ est $A \bowtie B = \{t \in D^{A \cup B} \mid t|_W \in A \ \wedge \ t|_U \in B\}$. Une séquence est une famille indexée par un préfixe de $\mathbb{N}$. On note $|$ le constructeur de séquence et $[]$ la séquence vide. On utilise la notation $a\,?b\,:c$ pour exprimer *si a alors b sinon c*.

Contraintes et CSP. Une *contrainte* $c = (W, T)$ est un couple composé d'un sous-ensemble $W \subseteq V$ de variables et d'une relation $T \subseteq D^W$ (W et T sont aussi respectivement notés $var(c)$ et $sol(c)$). Une contrainte vide (telle que $sol(c) = \emptyset$) est *fausse* et une contrainte pleine est telle que $sol(c) = D^W$. Quand $W = \emptyset$, seules ces deux contraintes existent : $(\emptyset, \emptyset)$ qui a pour valeur *faux* et $(\emptyset, ())$ qui a pour valeur *vrai*.
Un *problème de satisfaction de contraintes* (CSP) est un ensemble de contraintes. On note $var(C) = \bigcup_{c \in C} var(c)$ l'ensemble de ses variables et $sol(C) = \bowtie_{c \in C} sol(c)$ l'ensemble de ses solutions. Le CSP vide – ne contenant aucune contrainte– est *vrai* et est noté $\top$, tandis que tout CSP contenant une contrainte fausse est lui-même *faux* et noté $\bot$.

Préfixe et QCSP. Un *ensemble de variables quantifié*, ou *qset* est un couple (q, W) où $q \in \{\exists, \forall\}$ est un quantificateur et $W \subseteq V$.

Definition 1 (Préfixe) *Un A* préfixe P *est une séquence de qsets* $[(q_0, W_0), \ldots, (q_{n-1}, W_{n-1})]$ *tel que* $i \neq j \Rightarrow W_i \cap W_j = \emptyset$.

On note $P|_W$ la restriction du préfixe P aux variables d'un ensemble W. Une variable X est *déclarée* dans un qset W_i si $X \in W_i$. Un QCSP est défini en ajoutant un CSP à un préfixe :

Definition 2 (QCSP) *Un* CSP quantifié*, ou* QCSP *est un couple* (P, G) *où* P *est un préfixe et* G *un CSP appelé* goal.

Soit $P = [(q_0, W_0), \ldots, (q_{n-1}, W_{n-1})]$ un préfixe. Définissons les notations suivantes :

Premièrement, soit $range(P) = [0..n]$. Pour tout i de $range(P)$, soit $var_i(P) = W_i$ l'ensemble des variables déclarées à l'indice i, soit $before_i(P) = \bigcup_{j \leq i} var_j(P)$ (resp. $after_i(P) = before_n(P) \backslash before_i(P)$) l'ensemble des variables définies avant (resp. après) l'indice i.

Nous devons aussi pouvoir accéder à l'indice du prochain bloc universel $nu_i(P)$ situé après l'indice i. Définissons $nu_i(P) = min_{j>i}\{j \mid q_j = \forall\}$ si un tel indice existe, n sinon. Ces notations sont naturellement et directement étendues aux QCSP $Q = (P, G)$. De plus, on a $prefix(Q) = P$ et $goal(Q) = G$. Le QCSP est dit *clos* si $var(G) = before_n(Q)$, c'est-à-dire si toutes les variables mentionnées dans le goal sont explicitement quantifiées. Par la suite, nous considérerons uniquement des QCSP clos.

Example 3 (QCSP) *La formule :*

$$\exists X \in \{0,1\},\ \forall Y \in \{0,1\},\ \exists Z \in \{1,2\}\ .\ X + Y = Z$$

est représentée par le QCSP suivant (les domaines des variables n'étant pas mentionnés) :

$$Q = ([(\exists, X), (\forall, Y), (\exists, Z)], \{X + Y = Z\})$$

Ainsi, $prefix(Q) = [(\exists, X), (\forall, Y), (\exists, Z)]$, $goal(Q) = \{X + Y = Z\}$, $range(Q) = [1..3]$, $var_1(Q) = \{X\}$, $before_2(Q) = \{X, Y\}$, $after_2(Q) = \{Z\}$. □

Stratégie et scénario. La notion de solution d'un QCSP ne peut pas être une simple affectation de toutes ses variables comme pour un CSP. Elle doit être un ensemble d'affectations compatible avec la quantification universelle de certaines des variables. Intuitivement, une solution, appelée *stratégie*, contient la façon dont le joueur existentiel réagit à tous les coups possibles du joueur universel. Il est intéressant de noter qu'une stratégie est un objet syntactique indépendant de toute notion de validité : il s'agit juste d'un moyen possible de jouer au jeu comme s'il n'y avait pas de règle. Par conséquent, elle peut être définie uniquement sur un préfixe. Dans [13], une stratégie était définie comme une famille de fonctions (de Skolem) donnant chacune une valeur à une variable existentielle en fonction des universelles précédentes. Pour exposer cette notion d'un point de vue ensembliste, nous définissons plutôt une stratégie en extension, comme étant un ensemble de n-uplets. chacun de ces n-uplets est un *scénario*, c'est-à-dire un déroulement possible du jeu. La définition (inductive) de l'ensemble des stratégies pour un préfixe donné est la suivante :

Definition 4 (ensemble des stratégies) *L'*ensemble des stratégies $Strat(P)$ *d'un préfixe* $P = [(q_0, W_0), \ldots, (q_{n-1}, W_{n-1})]$ *est définie inductivement comme suit :*

- $Strat([]) = \emptyset$
- $Strat([(\exists, W) \mid P']) = \{t \bowtie s' \mid t \in D^W \wedge s' \in Strat(P')\}$
- $Strat([(\forall, W)|P']) = \{ \bigcup \alpha(D^W) \mid \alpha \in \Pi_{t \in D^W}(\{t \bowtie s' \mid s' \in Strat(P')\}) \}$

L'ensemble des stratégies pour un préfixe commençant par une variable universelle se définit ainsi : on construit, pour un n-uplet $t \in D^W$, l'ensemble $\{ t \bowtie s' \mid s' \in Strat(P') \}$ de toutes les stratégies commençant par t, et on prend le produit cartésien $\Pi_{t \in D^W}(\{ t \bowtie s' \mid s' \in Strat(P') \})$ de tous ces ensembles. Chaque n-uplet α de ce produit cartésien est aussi une fonction associant à chaque n-uplet de D^W une stratégie, qui est un ensemble de n-uplets. L'union des stratégies de l'ensemble image $\alpha(D^W)$ de cette fonction est une nouvelle stratégie contenant une sous-stratégie pour chaque $t \in D^W$. L'ensemble des stratégies pour le préfixe est l'ensemble de toutes les stratégies construites par tous les n-uplets α.

Sémantique d'un QCSP. Une stratégie est *gagnante* si tous ses scénarios satisfont le goal :

Definition 5 (Stratégie gagnante d'un QCSP) *Une stratégie s est une* stratégie gagnante *si, et seulement si $s|_{var(G)} \subseteq sol(G)$.*

On note $\text{WIN}(Q)$ l'ensemble de toutes les stratégies gagnante de Q.

Definition 6 (Sémantique d'un QCSP) *La* sémantique $[\![Q]\!]$ *d'un QCSP Q est :*

$$[\![Q]\!] = Win(Q)$$

Cette notion de solution généralise exactement la notion classique d'un CSP : un QCSP est vrai si et seulement si il admet une stratégie gagnante. D'autres notions plus faibles ont été proposées. [13] a utilisé la notion d'*outcome*, qui est l'ensemble des scénarios de toutes les stratégies gagnantes, comme notion de solution d'un QCSP pour modéliser le filtrage.

Example 7 *Considérons les QCSP suivants :*

$$\begin{array}{lll} Q_1: & \forall x \in \{0,1\}, \exists y \in \{1\}, \exists z \in \{0,1\}. & x \vee y = z \\ Q_2: & \exists x \in \{0,1\}, \forall y \in \{1\}, \forall z \in \{1\}. & x \vee y = z \\ Q_3: & \exists x \in \{0,1\}, \forall y \in \{0,1\}, \exists z \in \{1\}. & x \vee y = z \end{array}$$

On a alors :

$$\begin{array}{lcl} [\![Q_1]\!] & = & \{\ \{(0,1,1),(1,1,1)\}\ \} \\ [\![Q_2]\!] & = & \{\ \{(0,1,1)\},\ \{(1,1,1)\}\ \} \\ [\![Q_3]\!] & = & \{\ \{(1,0,1),(1,1,1)\}\ \} \end{array}$$

QCSP$^+$. Introduire la quantification restreints dans les QCSP passe par une modification de la nature du préfixe. en plus d'un quantificateur et d'un ensemble de variables, on ajoute un CSP dont les solutions définissent les valeurs autorisées pour les variables du qset courant. Les QCSP$^+$ ont été définis dans [3], essentiellement pour pallier à des problèmes de modélisation. Un *ensemble de variables quantifiées de manière restreinte*, ou *rqset* est un triplet (q, W, C) où $q \in \{\exists, \forall\}$ est un quantificateur, $W \subseteq V$ et C un CSP. Le but est de restreindre les valeurs possibles des variables de W à celles qui satisfont le CSP C. Nous étendons la notion de préfixe à ces rqsets : en particulier, $i \neq j \Rightarrow W_i \cap W_j = \emptyset$ doit toujours être vérifié.

Definition 8 (QCSP$^+$) *Un QCSP$^+$ est un couple $Q = (P, G)$ où P est un préfixe de rqsets tel que $var(C_i) \cap after_i(Q) = \emptyset$ et G est un CSP goal.*

Un QCSP$^+$ $Q = (P, G)$ est clos si $\forall i \in range(P), var(C_i) \subseteq before_i(Q)$ et $var(G) \subseteq before_n(Q)$. On remarque aisément qu'un QCSP standard est un QCSP$^+$ où $\forall i \in range(P), C_i = \emptyset$. La définition d'une stratégie d'un QCSP$^+$ est la même que pour un QCSP. Par contre, la notion de stratégie *gagnante* est différente : une stratégie gagnante est une stratégie pour laquelle tous les coups possibles du joueur universel mènent à un scénario gagnant. Comme dans les QCSP classiques, cela peut arriver quand toutes les contraintes du goal et des restrictions sont satisfaites. Mais cela peut aussi arriver quand la partie gauche d'une implication est fausse : ce scénario est alors valide quelles que soient les affectations des variables situées avec le rqset en question.

L'ensemble des stratégies gagnantes d'un QCSP$^+$ peut aussi être défini récursivement de la manière suivante :

Definition 9 (Stratégies gagnantes d'un QCSP$^+$) *Soit Q un QCSP$^+$. L'*ensemble des stratégies gagnantes $\text{WIN}(Q)$ *est défini par :*

- $\text{WIN}((\,[\,], G)) = sol(G)$
- $\text{WIN}((\,[(\exists, W, C)|P'], G)) = \{t \bowtie s \mid t \in D^W \wedge t|_{var(C)} \in sol(C) \wedge s \in \text{WIN}(P', G)\}$
- $\text{WIN}((\,[(\forall, W, C)|P'], G)) = \{\ \bigcup \alpha(D^W) \mid \alpha \in \Pi_{t \in D^W}(\ \{\ t \bowtie s \mid t|_{var(C)} \in sol(C)\ ?\ s \in \text{WIN}(P', G) :\ s \in \text{STRAT}(P', G)\ \}\ )\ \}$

Cette définition est analogue à la définition de l'ensemble des stratégies pour un préfixe, mais les sous-stratégies gagnantes ne sont pas les seules à être utiles, étant donné qu'une stratégie peut être gagnante à un niveau universel si elle contredit la restriction du niveau en question. Dès lors, n'importe quelle sous-stratégie, qu'elle soit gagnante ou non, peut être attachée.

La définition de la sémantique d'un QCSP s'applique aussi aux QCSP$^+$. Une méthode de propagation dans les QCSP+, appelée propagation en cascade, est décrite dans [3].

3 Optimisation

Les QCOP$^+$ sont créés à partir des QCSP$^+$ en ajoutant des fonctions de préférence et des agrégats sur les rqsets. Soit $\mathcal{A}$ un ensemble de noms d'agrégats et $\mathcal{F}$ un ensemble de fonctions d'agrégats. On définit une fonction d'agrégat f comme étant une fonction associant une valeur à un multi-ensemble, et dotée d'un élément neutre 0_f indiquant la valeur de $f(\{\!\{\}\!\})$. *somme, produit, moyenne, écart-type, médiane, cardinalité*,etc. sont des exemples de telles fonctions. Un *agrégat* est un atome de la forme $a : f(X)$ où $a \in \mathcal{A}, f \in \mathcal{F}$ et $X \in V \cup \mathcal{A}$. On appelle $names(A)$ l'ensemble des noms d'agrégats associé à un ensemble d'agrégats A. Une *condition d'optimisation* est un atome de la forme $\min(X), \max(X)$ où $X \in V \cup \mathcal{A}$, ou l'atome *any*. Un atome $\min(X)$ indique que l'on s'intéresse aux stratégies qui minimisent X et pas aux autres, tandis que

any indique l'absence de préférence sur les stratégies retournées. $max(X)$ est équivalent à $min(-X)$.

Definition 10 (Orqset) *Un* $\exists$-orqset *est un 4-uple* $(\exists, W, C, o)$ *où* $(\exists, W, C)$ *est un rqset et o une condition d'optimisation. Un* $\forall$-orqset *est un 4-uple* $(\forall, W, C, A)$ *où* $(\exists, W, C)$ *est un rqset et A un ensemble d'agrégats. Un* orqset *est soit un* $\exists$*-orqset soit un* $\forall$*-orqset.*

Nous étendons la notion de préfixe, ainsi que toutes les notations associées, à une séquence d'orqsets. Une restriction sur les variables possibles d'une condition d'optimisation ou d'un agrégat peut cependant apparaître : une variable à optimiser doit en effet avoir une unique valeur dans la stratégie courante. C'est effectivement le cas si il n'y a pas de $\forall$-orqset entre la condition d'optimisation et la définition de la variable à optimiser. Il est cependant possible d'optimiser sur un agrégat défini exactement au prochain bloc universel, étant donné que là encore, nous obtiendrons une valeur unique. Il en est de même pour une variable à agréger : celle-ci peut appartenir à l'ensemble des variables de n'importe quel bloc existentiel situé entre le bloc où l'agrégat est déclaré et le bloc universel suivant.

Considérons par exemple la séquence d'orqset suivante : $(\exists, W, C, o)$, $(\exists, W_2, C_2, o_2)$, $(\forall, W_3, C_3, A)$. La condition d'optimisation o peut porter sur les variables de W ou de W_2, ou encore sur une valeur d'agrégat obtenu à partir d'un élément de A. en effet, dans une sous-stratégie correspondante à cette séquence, ces variables ont une unique valeur. en revanche, o ne peut pas porter sur une des variables de W_3 étant donné que celles-ci prennent toutes les valeurs possibles dans la sous-stratégie.

Definition 11 (QCOP et QCOP$^+$) *Un QCOP*$^+$ *est un couple* (P, G) *où* G *est un CSP et* $P = [orq_0, \ldots, orq_{n-1}]$ *un préfixe d'orqsets tel que* $\forall i \in range(P)$*, avec* $k = nu_i(P)$ *:*

- *si* $orq_i = (\exists, W, C, o)$ *avec* $o = min(X)$ *ou* $o = max(X)$*, alors on doit avoir* $X \in before_{k-1}(P) \cup (k < n \ ? \ names(A_k) : \emptyset)$
- *si* $orq_i = (\forall, W, C, A)$*, alors pour tout* $a : f(X)$ *in A, on doit avoir* $X \in before_{k-1}(P) \cup (k < n \ ? \ names(A_k) : \emptyset)$

Un QCOP est un QCOP$^+$ *dans lequel aucun orqset ne possède de restriction.*

La sémantique d'un QCOP$^+$ est définie comme un ensemble de stratégies incluant le calcul des agrégats et respectant les conditions d'optimisation. Définissons pour commencer la fonction *val* qui calcule la valeur d'un agrégat $a{:}f(X)$ pour une stratégie s donnée. On a $val(a{:}f(X), s) = f(\{\!\{t|_X \mid t \in s\}\!\})$.

Definition 12 (Sémantique d'un QCOP$^+$) *La sémantique d'un QCOP*$^+$ *est un ensemble de stratégies tel que :*

- $\text{WIN}(([], G)) = sol(G)$
- $\text{WIN}(([(\exists, W, C, any)|P'], G)) = \text{WIN}(([(\exists, W, C)|P'], G))$
- $\text{WIN}(([(\exists, W, C, min(X))|P'], G)) = \{s \in \text{WIN}(([(\exists, W, C)|P'], G)) \mid s|_X = \min_{s' \in \text{WIN}(([(\exists, W, C)|P'], G))}(s'|_X)\}$
- $\text{WIN}(([(\forall, W, C, A)|P'], G)) = \{(val(a : f(X), s))_{a \in names(A)} \bowtie s \mid s \in \text{WIN}(([(\forall, W, C, A)|P'], G))\}$

Une fois les agrégats calculés, leurs valeurs sont attachés aux scénarios de la stratégie et ils apparaissent comme s'ils étaient des variables existentielles du niveau suivant. Tout comme un CSP peut avoir plusieurs solutions optimales, un QCOP$^+$ peut avoir plusieurs stratégies optimales. Cela peut se produire non seulement quand on utilise *any*, mais aussi quand plusieurs stratégies ont la même valeur optimale. Les sous-stratégies (ainsi que leurs valeurs optimales) peuvent varier énormément d'une stratégie optimale à l'autre. Cependant, l'algorithme de recherche décrit dans la section suivante retourne une seule de ces stratégies optimales.

4 Algorithmes

Cette section présente un algorithme de recherche évaluant les QCOP$^+$ et esquisse une amélioration basée sur le principe du branch-and-bound. Cet algorithme de recherche a été implémenté dans le solveur QeCode [10], basé sur Gecode [11]. Cette technique de résolution est basée sur la procédure de recherche des QCSP$^+$ qui explore la structure de quantification du problème de manière récursive. Un mécanisme d'extraction de la stratégie et son stockage sous forme d'arbre a été ajouté. Cette dernière fonctionnalité améliore aussi la réponse donnée à un QCSP$^+$, étant donné que l'utilisateur ne s'intéresse généralement pas au problème de décision lui même mais aussi à la manière précise de "gagner au jeu". Une représentation explicite des stratégies – que [12] appelle certificats – possède de nombreuses application, la première étant de pouvoir vérifier la solution d'une manière indépendante du solveur. Actuellement, la stratégie est stockée de manière non compressée, ce qui peut constituer une limite à la taille des instances pouvant être résolues.

La procédure de recherche principale se compose de deux fonctions d'évaluation mutuellement récursives, l'une traitant les $\exists$-orqset et l'autre dédiée aux $\forall$-orqset. Toutes deux retournent une stratégie sous forme d'un arbre qui peut être soit l'arbre vide *null* soit *tree(a,B)* où a est un n-uplet et B un ensemble

d'arbres. La figure 4 montre ces trois algorithmes. Pour un ∃-orqset, la fonction garde en mémoire la meilleurs stratégie rencontrée jusque là (*BEST_STR*) et la retourne, ou renvoie null si le sous-problème est faux. Toutes les stratégies sont successivement explorées et leur valeur de X comparées. Les conditions max et *any* sont traitées de manière similaire.

```
Procedure Solve ([o|P'], G)
  if o = orqset existentiel then
    return Solve_e ([o|P'], G)
  else
    return Solve_u ([o|P'], G)
  end if

Procedure Solve_e([(∃, W, C, min(X))|P'], G)
  BEST_STR := null
  BEST_Xvalue := +∞
  for all t ∈ D^W t.q. t solution de C do
    CUR_STR := Solve( (P', G)[W ← t] )
    if CUR_STR ≠ null then
      CUR_Xvalue := CUR_STR|_X
      if CUR_Xvalue < BEST_Xvalue
      then
        BEST_STR := CUR_STR
        BEST_Xvalue := CUR_Xvalue
      end if
    end if
  end for
  return tree(t,{ BEST_STR })

Procedure Solve_u ([(∀, W, C, A)|P'], G)
  for all a:f(X) ∈ A do
    VAL_a := ∅
  end for
  STR := ∅
  for all t ∈ D^W t.q. t solution de C do
    CUR_STR := Solve( (P', G)[W ← t] )
    if CUR_STR = null then
      return null
    else
      for all a:f(X) ∈ A do
        VAL_a := VAL_a ∪
          {{ CUR_STR|_X }}
      end for
      STR := STR ⋃ CUR_STR
    end if
  end for
  return tree((f(VAL_a))_{a:f(X)∈A}, STR)
```

FIG. 4 – Procédure de recherche.

La forme la plus simple de branch-and-bound applicable à cette procédure de recherche consiste à simplement poster la contrainte $X < BEST\_Xvalue$ (resp. $>$) aux branches restantes à explorer du problème de minimisation (resp. maximisation). Il s'agit d'une adaptation directe de l'algorithme de [14] pour lequel les bornes inférieure/supérieure sont fixées par la condition d'optimisation et associées à sa variable d'optimisation. Quand une solution est trouvée, la partie de l'algorithme dédiée aux ∀-orqsets évalue les sous-stratégies découlant de chaque n-uplet valide pour cet orqset, calcule les agrégats et finalement retourne l'ensemble *STR* de ces sous-stratégies.

Branch and Bound. Cet algorithme de Branch-and-Bound est incorrect dans le cas où plusieurs conditions d'optimisations se chevauchent, c'est-à-dire si il existe deux orqsets $orq_i = (\exists, W_i, C_i, min(X))$ avec $X \in W_k$ et $orq_j = (\exists, W_j, C_j, min(Y))$ avec $Y \in W_l$ tels que $i < j < k$.

$$
\begin{array}{l}
\exists X \in D_X \\
\quad \exists Y \in D_Y \\
\quad\quad \exists A \in D_A \\
\quad\quad\quad \exists B \in D_B \\
\quad\quad\quad\quad \ldots \\
\quad\quad\quad any \\
\quad\quad any \\
\quad \min(B) \\
\min(A)
\end{array}
$$

FIG. 5 – Cas où le B&B est incorrect

Example 13 *Un exemple de problème où le branch-and-bound est incorrect est donnéen figure 5. Supposons qu'il existe trois stratégies* $s0 = \{(X_0, Y_0, A_0, B_0)\}$, $s_1 = \{(X_1, Y_1, A_1, B_1)\}$ *et* $s_2 = \{(X_1, Y_2, A_2, B_2)\}$ *telles que* $A_1 > A_0$, $A_2 < A_0$ *et* $B_1 < B_2$. *Une fois* s_0 *trouvée, la contrainte* $A < A_0$ *est ajoutée à la recherche des stratégies suivantes. Et donc,* s_1 *n'est plus considérée, et on trouve* s_2 *dont la valeur* A_2 *pour* A *est meilleure, ce qui en fait la stratégie optimale retournée. Or, sans l'application du branch-and-bound, la condition d'optimisation au niveau de* Y *aurait retourné la stratégie* s_1, *sa valeur de* B *étant meilleure que celle de* s_2, *et donc la valeur* A_1 *aurait été retournée au niveau supérieur, et la meilleure stratégie au niveau supérieur aurait donc bien été* s_0.

Proposition 14 *Le branch-and-bound est correct aux niveaux où des conditions d'optimisations ne se chevauchent pas.*

Idée de la preuve En utilisant les mêmes notations que ci-dessus, les conditions d'optimisations ne se chevauchent pas si $k \leq j$. Dès lors, n'importe quelle branche coupée par le branch-and-bound le sera avant que l'on atteigne le niveau de Y, et donc, seules des stratégies moins bonnes pour X seront supprimées. □

5 Examples

Dans cette section, nous donnons plusieurs exemples de modélisation de problèmes utilisant l'optimisation,

allant des exemples jouets à des problèmes réels pris dans la littérature sur la programmation bi-niveaux. Nous utilisons une syntaxe dans laquelle les agrégats et les conditions d'optimisations apparaissent à la fin, de manière à rendre les problèmes plus lisibles. Par exemple, voici un QCOP$^+$ qui retourne une stratégie dans laquelle $X = 0$ si la somme des indices impairs du tableau A est inférieur à la somme de ses indices pairs et $X = 1$ sinon, présenté de manière formelle, puis dans cette syntaxe :

$$
\begin{array}{l}
(\ [\ (\exists, \{X\}, \emptyset, min(s)), \\
\quad (\forall, \{i\}, \{i \bmod 2 = X\}, \{s : sum(Z)\}), \\
\quad\quad (\exists, \{Z\}, \emptyset, any) \ ], \\
\quad\quad\quad \{Z = A[i]\} \ )
\end{array}
$$

```
const A[0..9]
∃ X in 0..1
| ∀ i in 0..9 [i mod 2 = X]
| | ∃ Z in 0..+oo
| | | Z = A[i]
| | any
| s :sum(Z)
min(s)
```

Ordonnancement avec adversaire et minimax. Souvent, les objectifs de deux agents sont strictement opposés. Cette situation peut être résolue par un algorithme minimax classique. Dans ce cas, le branch-and-bound est équivalent au filtrage alpha-beta. Nus illustrons ce cas avec un extension de l'exemple de l'ordonnancement avec adversaire présenté dans [4]. Ce problème implique deux opposants : l'*ordonnanceur* qui cherche à établir un plan respectant des contraintes données (ressources disponibles et temps imparti), tandis qu'un *adversaire* cherche à empêcher l'établissement d'un tel plan en modifiant (dans certaines limites) les données du problème original. Dans la version en QCSP$^+$ de ce problème, l'ordonnanceur cherchait à établir un plan se terminant avant une date limite donnée. Il est maintenant possible de chercher à minimiser ce temps pour une attaque donnée, tandis que l'adversaire cherche à maximiser ce même temps. Considérons par exemple trois tâches a_1, a_2, a_3 et une ressource r. Chaque tâche a_i a une date de début s_i, une durée d_i et demande c_i unités de la ressource r, dont la capacité maximale est 5. L'ordre de précédence des tâches est $a_1 \prec a_2$, et les données du problème sont $d_1 = 1, d_2 = 2, d_3 = 3, c_1 = 3, c_2 = 2$ et $c_3 = 1$. L'adversaire est capable d'augmenter la consommation de ressource d'au plus deux tâches d'une unité. Nous ajoutons une tâche factice *end* qui servira à minimiser la durée totale du plan. Le QCOP$^+$ est le suivant :

```
const d[1..3], c[1..n]
∃ k1 ∈ 0,1, k2 ∈ 0,1, k3 ∈ 0,1
| [k1+k2+k3 =< 2]
| ∃ S1 ∈ D1, S2 ∈ D2, S3 ∈ D3, Send ∈ Dend,
| | c'1 ∈ Dc1, c'2 ∈ Dc2, c'3 ∈ Dc3
| | [S1+d1 =< Send, S2+d2 =< Send, S3+d3 =< Send,
| | S1+D1 =< S2, c'1=c1+k1, c'2=c2+k2, c'3=c3+k3]
| | cumulative([S1,S2,S3], [d1,d2,d3], [c'1,c'2,c'3], 5)
| minimize(Send)
maximize(Send)
```

Étant donné qu'il n'y a que des variables existentielles, la stratégie est réduite à un tronc donnant l'attaque la plus puissante et la réponse de l'ordonnanceur.

Tarification de liens. Voici un exemple venant de l'industrie des télécoms, repris de [15].

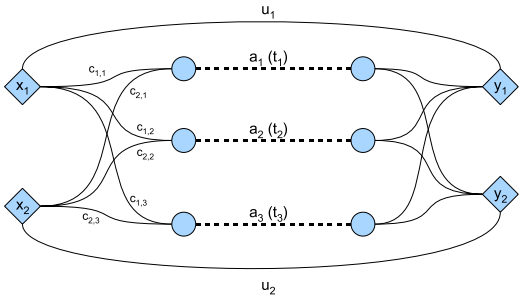


FIG. 6 – Un problème de tarification de liens

Le problème consiste à définir un tarif d'utilisation de plusieurs liens de manière à maximiser le bénéfice de leur propriétaire (le meneur). Le réseau est décrit en figure 6. il se compose de $NCustomer$ clients (les suiveurs)qui cherchent à acheminer leurs données à moindre coût, indépendamment les uns des autres. Chaque chemin d'une source vers une destination doit passer par un arc a_j, sachant que le chemin de s_i à a_j est taxé au client à hauteur de c_{ij} par un autre fournisseur d'accès. On considère que chaque client i cherche à minimiser ses propres dépenses, et qu'il peut toujours choisir une offre concurrente qui lui coûtera u_i. Le but du problème est de déterminer les tarifs t_j à appliquer de manière à maximiser le bénéfice de l'opérateur téléphonique. La figure 6 montre cette situation avec 2 clients et 3 liens. Ce problème peut s'exprimer en QCOP$^+$ comme suit :

```
const NCustomer
const NArc
// c[i,j] = coût pour aller de Ci à Aj
const c[NCustomer,NArc]
// d[i] = quantités de données du client i
const d[NCustomer]
// u[i] = prix maximum pour le client i
const u[NCustomer]
∃ t[1], ..., t[NArc] ∈ [0,max]
| ∀ k ∈ [1,NCustomer]
```

```
| | ∃ a ∈ [1,NArc],
| | | cost ∈ [1,max],
| | | income ∈ [0,max]
| | | cost = (c[k,a]+t[a])*d[k]
| | | income = t[a]*d[k]
| | | cost =< u[k]
| | minimize(cost)
| s :sum(income)
maximize(s)
```

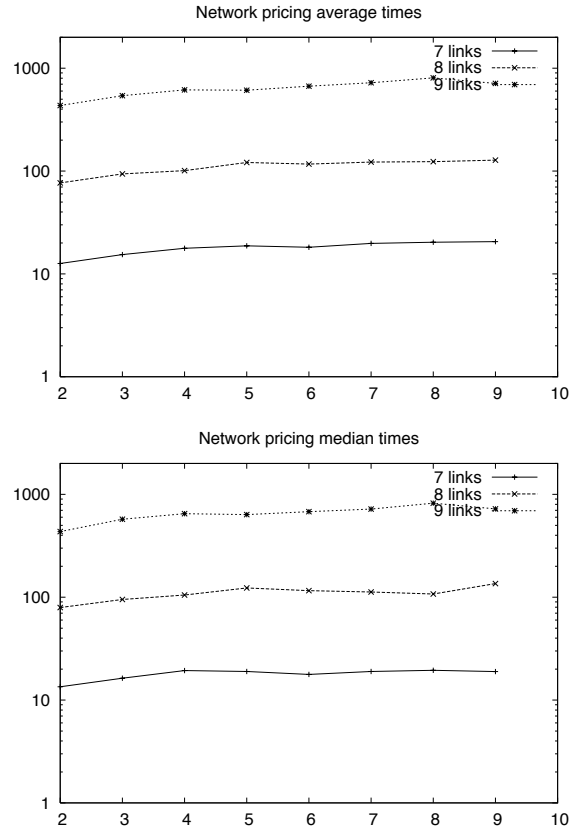


FIG. 7 – moyenne et médiane des temps de résolution (ordonnée, en secondes) du *problème de tarification de liens* pour 7, 8 et 9 liens, et pour entre 2 et 9 clients(en abscisse).

Nous avons généré aléatoirement des ensembles d'instances de ce problème. Ces ensembles diffèrent les uns des autres suivant deux paramètres : d'une part, le nombre de liens détenus par l'opérateur et d'autre part le nombre de clients voulant utiliser ces liens. L'opérateur peut choisir entre 5 tarifs possibles pour chacun de ses liens. Dans un ensemble donné, chaque instance varie sur le prix maximum acceptable par chaque client, ainsi que sur les coûts initiaux pour acheminer les données du point de départ vers le point d'entrée de chaque lien, ces données étant tirées aléatoirement. Chaque ensemble contient 100 instances.

Ces tests ont été menés sur des machines équipées chacune de deux Opteron dual-core et de 4 Go de RAM. QeCode étant mono-threadé, chacun des coeurs d'exécution traitait une instance différente. La résolution de toutes les instances a été menée à terme. La figure 7 montre les temps de résolution moyens et médians de ces tests. Les instances ayant un nombre de liens inférieur à 7 ne sont pas montrées car la majorité d'entre elles sont résolues en moins d'une seconde. On remarque que le nombre de clients influe peu sur le temps de résolution par rapport au nombre de liens. Ceci est naturel, étant donné que l'ajout d'un client conduit simplement à déterminer quel chemin il empruntera, alors que l'ajout d'un lien offre un choix de plus à chaque client et, surtout, multiplie les différents choix possibles de tarification de l'opérateur.

Tarification de réseaux virtuels. Les infrastructures des réseaux télécoms sont extrêmement coûteuses à mettre en place. de ce fait, seules une poignée d'*opérateurs télécom* (NO) les détiennent. De manière à créer de la concurrence, les gouvernement ont soutenu la mise en place d'*opérateurs* (VNO) qui fournissent les mêmes services bien qu'ils ne possèdent pas leur propre infrastructure, et louent de la bande passante au réseau d'un NO à la place. Dans un tel environnement économique, les décisions nécessitent un modèle d'oligopole complexe et loin des règles de l'équilibre général et de la concurrence parfaite de Walras. Les acteurs sont en concurrence tout en coopérant, dans des limites fixées par l'autorité de régulation. Ces exemple est tiré de [16].

La figure 8 représente les relations entre le NO, le VNO et les clients, chacun des acteurs étant modélisé dans [16]. On se place du point de vie du NO et l'objectif principal du modèle est de déterminer les décisions $y = (y_1, y_2)$, y_1 étant la fourniture de service à ses propres clients et y_2 le prix de location au VNO. Les décisions prises par le VNO sont $z = (z_1, z_2)$, z_1 étant le tarif du service à ses clients et z_2 la capacité louée au NO. Les clients sont modélisés par $n = (n_1, n_2)$ ((nombre total de clients , respectivement du NO et du VNO) en fonction des tarifs

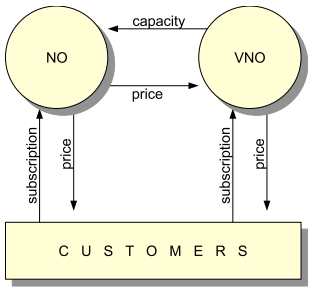


FIG. 8 – Tarification de réseaux virtuels

fixés : $n_i = k_i + r_{i,1}y_1 + r_{i,2}y_2$, les paramètres $k_i, r_{i,1}$ and $r_{i,2}$ étant déterminés par l'analyse du marché. Le bénéfice du VNO est égal au revenu tiré de ses clients moins le tarif de location au NO, c'est à dire $(q - e_2z_1)n_2 - y_2z_2 - g_2$, où q, e_2 et g_2 sont respectivement le coût variable par client et le coût fixe de fourniture de service. Notons que le revenu du NO dépend des décisions proses par le VNO. De plus, le tarif appliqué au client pour c service est compris entre des limites d et D. Le prix de location a une borne supérieure U_1 fixée par l'autorité de régulation et la capacité maximale disponible pour le VNOest inférieurs à une limite U_2 donnée.

Cela nous amène au au modèle suivant. Le quantificateur universel n'est pas utilisé, étant donné qu'il n'y a qu'un seul opérateur virtuel. Là encore, les conditions d'optimisation se chevauchent et il est donc incorrect d'utiliser le branch-and-bound.

```
const d, D, U1, U2, k1, r11, r12, r21, r22, g1, q
∃ y1 ∈ Dy1, y2 ∈ Dy2
| [d =< y1, y1 =< D, y2 =< U1]
| ∃ z1 ∈ Dz1, z2 ∈ Dz2
| | [d =< z1, z1 =< D, z2 =< U2]
| | ∃ n1 ∈ Dn1, n2 i∈ Dn2, rno ∈ Drno, rvno ∈ Drvno
| | | n1 = k1 - r11 * y1 + r12 * z1
| | | n2 = k2 + r21 * y1 - r22 * z1
| | | rvno = (q - e2 * z1) * n2 - y2 * z2 - g2
| | | rno = g1 + (q + y1 + e1) * n1 + y2 * z2
| maximize(rvno)
maximize(rno)
```

6 Discussion

On peut se demander quelle est la classe de complexité des QCOP$^+$. Cette question mériterai plus de travail théorique pour décrire l'analogue de la classe NPO pour les problèmes PSPACE.

En plus de constituer des conditions d'optimisation, min et max sont aussi des fonctions d'agrégat possibles. Il y a cependant une différence entre l'agrégat min (que l'on appelle min-agg) et la condition d'optimisation (que l'on appelle min-opt dans le sens où min-agg force toutes les branches de l'universelle à posséder des sous-stratégies alors que min-opt renvoie le minimum des branches non contredites. De plus, si le quantificateur restreint lui-même n'a pas de solution, min-agg renvoie $0_{\min}$ alors que min-opt échoue. Dans les exemples, on utilise le quantificateur existentiel autant pour le principal décideur que pour son adversaire car, le problème ne permettant pas à l'adversaire de faire complètement échouer le projet, il suffit de rechercher la stratégie optimale pour l'un comme pour l'autre.

Ce papier s'approche de plusieurs travaux : le formalisme *Plausibility-Feasibility-Utility* (PFU) [17], les *expressions itérées* [18] et les CSP stochastiques [14]. Le but du premier est de fournir une unification algébrique de plusieurs formalismes – les QBF, les QCSP les CSP stochastiques, les réseaux Bayésiens et les les processus de décision de Markov –, tandis que le deuxième cherche à modéliser l'allocation de ressource dans les workflows. Tous deux introduisent des expressions de la forme $\bigoplus_{x_1 \in D_1} \dots \bigoplus_{x_n \in D_n} expr(x_1, \dots x_n)$ avec $\bigoplus \in \{\min, \max, \sum, \Pi\}$. Il est impossible d'exprimer les modèles bi-niveaux dans ces formalismes car la condition d'optimisation doit s'appliquer sur le résultat d'une sous-expression immédiate. Cependant, des constructions comme $\min(\sum_x e_1(x) + \sum_y e_2(y))$ peuvent être exprimées par une PFU ou une expression itérée et pas par un QCOP$^+$. de plus, la condition pour que le branch-and-cound puisse s'appliquer est toujours vérifiée par construction.

[19] propose de trouver une stratégie d'un QCSP booléen qui maximise une somme pondérée ds variables existentielles ayant la valeur *vrai*. Il prouve un théorème de dichotomie permettant de classer approximativement de tels problèmes en classes praticables et impraticables. Tous ces travaux, y compris le langage des QCOP$^+$, posent le problème de trouver un langage adéquat pour exprimer le choix d'une stratégie dans les problèmes quantifiés. Quel est l'équivalent des CSP pondérés dans ce contexte ?

Quand plusieurs stratégies sont optimales au premier niveau, il peut arriver qu'elles diffèrent considérablement au niveau des sous-stratégies induites. Le langage des QCOP$^+$ ne permet pas pour le moment d'exprimer ce genre de "meta-optimisation". Un autre problème serait de permettre de poster des contraintes entre des variables et des résultats d'agrégat. Il est par exemple impossible d'exprimer le fait que les sous-stratégies doivent attribuer une valeur différente à une variable existentielle pour chaque valeur d'une variable universelle : cela nécessiterait une contrainte agrégat "Alldifferent" qui pourrait échouer. L'évaluation des stratégies partielles et/ou non-gagnantes reste aussi une question ouverte qui pourrait ouvrir la voie à l'adaptation de la relaxation de contraintes et à la recherche locale dans les problèmes quantifiés.

7 Conclusion

Nous avons présenté le formalisme des QCOP$^+$ pour modéliser des problèmes d'optimisation quantifiés. Ce cadre est suffisamment large pour englober les problèmes bi-niveaux (et multi-niveaux) qui sont étudiés en théorie des jeux et en recherche opérationnelle depuis des années. Les quantificateurs existentiels

peuvent être dotés d'une condition d'optimisation qui permet de choisir une stratégie optimisant un paramètre, et les quantificateurs universels sont utilisés pour calculer des agrégats. Ce travail fournit un moyen d'exprimer et de résoudre de tels problèmes et étends les QCSP à des problèmes plus généraux que l'étude du pire cas.

Ce travail est supporté par le projet ANR-06-BLAN-0383.

Références

[1] Nightingale, P. : Consistency for quantified constraint satisfaction problems. In van Beek, P., ed. : CP. Volume 3709 of Lecture Notes in Computer Science., Springer (2005) 792–796

[2] Bessière, C., Verger, G. : Strategic constraint satisfaction problems. In Miguel, I., Prestwich, S., eds. : Workshop on Constraint Modelling and Reformulation, Nantes, France (2006) 17–29

[3] Benedetti, M., Lallouet, A., Vautard, J. : QCSP Made Practical by Virtue of Restricted Quantification. In Veloso, M., ed. : International Joint Conference on Artificial Intelligence, Hyderabad, India, AAAI Press (2007) 38–43

[4] Benedetti, M., Lallouet, A., Vautard, J. : Modeling adversary scheduling with QCSP+. In : ACM Symposium on Applied Computing, Fortaleza, Brazil, ACM Press (2008)

[5] Nightingale, P. : Consistency and the Quantified Constraint Satisfaction Problem. PhD thesis, University of St Andrews (2007)

[6] Bracken, J., McGill, J. : Mathematical programs with optimization problems in the constraints. Operations Research **21** (1973) 37–44

[7] Stackelberg, H. : The theory of market economy. Oxford University Press (1952)

[8] Bialas, W.F. : Multilevel mathematical programming, an introduction. Slides (2002)

[9] Colson, B., Marcotte, P., Savard, G. : An overview of bilevel optimization. Annals of Operations Research **153** (2007) 235–256

[10] QeCode Team : QeCode : An open QCSP+ solver (2008) Available from `http ://www.univ-orleans.fr/lifo/software/qecode/`.

[11] Gecode Team : Gecode : Generic constraint development environment (2006) Available from `http ://www.gecode.org`.

[12] Benedetti, M. : Extracting certificates from quantified boolean formulas. In Kaelbling, L.P., Saffiotti, A., eds. : International Joint Conference on Artificial Intelligence, Edinburgh, Scotland, Professional Book Center (2005) 47–53

[13] Bordeaux, L., Cadoli, M., Mancini, T. : CSP properties for quantified constraints : Definitions and complexity. In Veloso, M.M., Kambhampati, S., eds. : National Conference on Artificial Intelligence, AAAI Press (2005) 360–365

[14] Walsh, T. : Stochastic constraint programming. In : ECAI. (2002) 111–115

[15] Bouhtou, M., Grigoriev, A., van Hoesel, S., van der Kraaij, A.F., Spieksma, F.C., Uetz, M. : Pricing bridges to cross a river. Naval Research Logistics **54**(4) (2007) 411–420

[16] Audestad, J.A., Gaivoronski, A.A., Werner, A. : Extending the stochastic programming framework for the modeling of several decision makers : pricing and competition in the telecommunication sector. Annals of Operations Research **142**(1) (2006) 19–39

[17] Pralet, C., Verfaillie, G., Schiex, T. : An algebraic graphical model for decision with uncertainties, feasibilities, and utilities. Journal of Artificial Intelligence Research **29** (2007) 421–489

[18] Bordeaux, L., Hamadi, Y., Quimper, C.G., Samulowitz, H. : Expressions Itérées en Programmation par Contraintes. In Fages, F., ed. : Journées Francophones de Programmationpar Contraintes. (2007) 98–107

[19] Chen, H., Pál, M. : Optimization, games, and quantified constraint satisfaction. In Fiala, J., Koubek, V., Kratochvíl, J., eds. : MFCS. Volume 3153 of Lecture Notes in Computer Science., Springer (2004) 239–250

Sémantique et calcul syntaxique pour les formules booléennes quantifiées

Igor Stéphan

LERIA, Université d'Angers, 2 Boulevard Lavoisier, 49045, Angers, Cedex 01, France
igor.stephan@info.univ-angers.fr

Résumé

Nous revisitons la sémantique des formules booléennes quantifiées et étudions une relation d'équivalence qui en préserve les modèles. Nous proposons un calcul syntaxique pour les formules booléennes quantifiées basé sur le concept de relation de formules et mettons en exergue le lien avec la relation d'équivalence étudiée prouvant ainsi la correction et la complétude.

1 Introduction

Le formalisme des formules booléennes quantifiées a été introduit initialement dans [18] puis largement approfondi dans [27]. A l'origine, ce formalisme qui décrit une « hiérarchie » de langages était un outil théorique pour explorer des classes de complexité au-delà de la célèbre classe NP mais il est devenu depuis [9, 10] un domaine de recherche plus appliqué qui a vu l'implantation de multiples procédures de décision et son utilisation dans de nombreux domaines de l'intelligence artificielle (par exemple en planification [22, 1] et en implantation de procédures de décision [5]) et dans la vérification formelle de circuit (voir [4] pour un florilège). La plus grande part des procédures de décision récentes pour la validité des QBF [10, 28, 17, 16] sont des extensions de l'algorithme de recherche de Davis, Logemann et Loveland [12] pour le problème de la satisfiabilité booléenne ; d'autres sont basées sur le principe de résolution [23] telle que la Q-resolution [7] ou Quantor [6] (qui combine cette dernière avec l'expansion) ; d'autres sont basées sur l'élimination des quantificateurs à la Fourier-Motzkin [20, 19] ; d'autres enfin, sans être exhaustif, sur la skolémisation [2]. La sémantique des QBF est généralement présentée soit dans sa forme décisionnelle, soit dans sa forme fonctionnelle grâce essentiellement aux fonctions de Skolem [8, 3] (des fonctions booléennes associées aux symboles existentiellement quantifiés de la formule dépendant des symboles universellement quantifiés qui les précèdent) qui peuvent être revisitées par exemple en des politiques [11]. La sémantique présentée sous sa forme décisionnelle, si elle est bien adaptée au problème théorique du test d'appartenance à un langage, ne permet pas d'extraire les solutions de la QBF et n'est donc pas suffisante lorsque l'on désire aider le « joueur existentiel » à gagner si l'on considère la QBF comme étant un jeu à deux joueurs (i.e. quels que soient les coups du « joueur universel », le « joueur existentiel » a un coup qui le mène à la victoire). Nous nous proposons de revisiter la définition de la sémantique des formules booléennes quantifiées pour en jeter des bases définies par induction permettant de traiter les symboles propositionnels libres, les symboles propositionnels existentiellement quantifiés definis par une solution et les symboles propositionnels existentiellement quantifiés qui ne sont pas définis dans une solution et qui sont éliminés suivant la sémantique habituelle basée sur la disjonction. Cette nouvelle définition de la sémantique nous permet d'explorer une relation d'équivalence basée non pas sur la préservation de la validité mais sur celle des solutions. Cette relation nous permet de définir un calcul syntaxique pour les formules booléennes quantifiées basé sur la notion de relation de formules [24].

2 Préliminaires

Les valeurs booléennes sont notées **vrai** et **faux**. L'ensemble des valeurs booléennes est noté **BOOL**. L'ensemble des symboles propositionnels est noté $\mathcal{SP}$. Le symbole $\wedge$ est utilisé pour la conjonction, $\vee$ pour la disjonction, $\neg$ pour la négation, $\rightarrow$ pour l'implication et $\leftrightarrow$ pour la bi-implication. L'ensemble des formules propositionnelles est dénoté **PROP**. Un littéral est

un symbole propositionnel ou la négation de celui-ci. Le complémentaire d'une formule propositionnelle F, noté $\overline{F}$, est G si $F = \neg G$ et $\neg F$ sinon. L'ensemble des sous-formules et leurs complémentaires d'une formule propositionelle F, incluant $\top$ et $\overline{\top}$, est dénoté $sub(F)$. Une formule propositionnelle est sous forme normale conjonctive (FNC) si c'est une conjonction de disjonctions de littéraux. Une substitution est une fonction de l'ensemble des symboles propositionnels dans l'ensemble **PROP**. Nous définissons la substitution d'un symbole propositionnel x par F dans G, notée $[x \leftarrow F](G)$, comme étant la formule obtenue de G en remplaçant toutes les occurrences du symbole propositionnel x par la formule F. Une valuation v est une fonction de $\mathcal{SP}$ dans **BOOL** et l'interprétation, notée v^* est son extension dans **PROP**. Le symbole $\exists$ est utilisé pour la quantification existentielle et $\forall$ pour la quantification universelle. Toute formule propositionnelle est aussi une formule booléenne quantifiée (QBF). Si F est une QBF et x est un symbole propositionnel alors $(\exists x\ F)$ et $(\forall x\ F)$ sont des QBF. Un lieur est une chaîne de caractères $q_1 x_1 \dots q_n x_n$ avec $x_1, \dots, x_n$ des symboles propositionnels distincts et $q_1 \dots q_n$ des quantificateurs ; le lieur vide est noté ε ; par convention, des quantificateurs différents lient des symboles propositionnels différents. Une QBF constituée d'un lieur et d'une formule booléenne appelée matrice est une QBF prénexe ; si tout symbole propositionnel apparaissant dans une QBF possède une occurrence dans le lieur elle est dite close. Nous nous restreignons par la suite aux QBF prénexes. Le lieur induit une relation d'ordre sur les symboles propositionnels qui est notée $<$ (plus un symbole est à gauche dans la lieur plus il est petit). Une QBF est en FNC si sa matrice l'est. La sémantique des symboles booléens est définie de manière habituelle ; en particulier, à chaque connecteur (resp. $\top$, $\bot$, $\neg$, $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$) est associée une fonction booléenne (resp. $i_\top, i_\bot :\rightarrow$ **BOOL** $i_\neg$: **BOOL** $\rightarrow$ **BOOL**, $i_\wedge, i_\vee, i_\rightarrow, i_\leftrightarrow$: **BOOL**$\times$**BOOL** $\rightarrow$ **BOOL**) qui en définit sa sémantique. A une fonction booléenne f d'arité n (i.e. une fonction de $\mathbf{BOOL}^n$ dans **BOOL**) est associée une formule propositionnelle ψ_f sur les symboles propositionnels $\{x_1, \dots, x_n\}$ telle que $v^*(\psi_f) =$ **vrai** si et seulement si $f(v(x_1), \dots, v(x_n)) =$ **vrai** pour toute valuation v.

3 Sémantique et relations d'équivalences

Nous revisitons la sémantique des QBF puis nous explorons les propriétés d'une relation d'équivalence non pas basée sur la préservation de la validité comme pour l'équivalence classique mais sur la préservation des modèles QBF. La sémantique d'une QBF prénexe en définit la valeur de vérité selon une *valuation QBF* composée d'une valuation propositionnelle et d'une *valuation fonctionnelle.*

Définition 1 (valuation QBF) *Une fonction partielle sk de l'ensemble des symboles propositionnels dans l'ensemble des fonctions booléennes est une* valuation fonctionnelle *pour une QBF prénexe si pour tout symbole propositionnel existentiellement quantifié x il existe un (unique) couple $(x \mapsto \hat{x}) \in sk$ tel que la fonction booléenne $\hat{x}$ a pour arité le nombre de symboles propositionnels universellement quantifiés qui précèdent x dans le lieur. L'ensemble des valuations fonctionnelles est noté* **VAL_FONC**. *Une valuation QBF est un couple constitué d'une valuation propositionnelle et d'une valuation fonctionnelle. L'ensemble des valuations QBF est noté* **VAL_QBF** $=$ **VAL_PROP** $\times$ **VAL_FONC**. *Une valuation QBF est* partielle *si tous les symboles propositionnels existentiellement quantifiés de la QBF prénexe ne sont pas présents dans la valuation fonctionnelle.*

Une valuation QBF qui n'est pas partielle peut être qualifiée, pour insister, de « valuation QBF totale ».

Exemple 1 *Soit la QBF $\xi = \forall a \exists b \forall c \exists d \mu$ avec $\mu = \neg((c \rightarrow b) \rightarrow \neg(d \rightarrow a))$. La fonction partielle $\{(b \mapsto \hat{b}), (d \mapsto \hat{d})\}$, $\hat{b}$ d'arité 1 et $\hat{d}$ d'arité 2, est une valuation fonctionnelle qui forme avec toute valuation propositionnelle une valuation QBF (totale) pour la QBF ξ ; les valuations fonctionnelles $\emptyset$, $\{(b \mapsto \hat{b})\}$ et $\{(d \mapsto \hat{d})\}$ en forment avec toute valuation propositionnelle des valuations QBF partielles.*

Une valuation fonctionnelle est un ensemble de fonctions booléennes qui sont très souvent appelées «fonctions de Skolem». Dans la littérature, la valuation propositionnelle est une fonction qui associe à tout symbole propositionnel une valeur de vérité ; pour la valuation fonctionnelle, une fonction partielle est préférée à une fonction totale.

La sémantique d'une QBF prénexe est définie pour des QBF prénexes telles que des quantificateurs différents sont associés à des symboles propositionnels différents. À chaque connecteur logique est associée une fonction à valeurs dans **BOOL** qui en définit sa sémantique au niveau propositionnel. Les définitions suivantes reprennent l'organisation de la définition de la sémantique de la logique des prédicats proposée dans [14] : une définition de la sémantique des connecteurs et constantes logiques au niveau propositionnel identique à celle de la sémantique de la logique propositionnelle, une définition de la sémantique des connecteurs, constantes et quantificateurs au niveau QBF comme une restriction de celle de la sémantique de la logique des prédicats, une définition de la sémantique des QBF comme extension aux formules de la

sémantique des connecteurs, constantes et quantificateurs. Nous définissons la sémantique au niveau QBF des connecteurs et constantes logiques ainsi que des quantificateurs.

Définition 2 (sémantique des connecteurs et quantificateurs) *La sémantique des connecteurs et quantificateurs est définie par (v une valuation propositionnelle et sk une valuation fonctionnelle) :*

$$I_\top, I_\bot : \mathbf{VAL\_QBF} \to \mathbf{BOOL}$$

$$I_\bot(v, sk) = i_\bot \quad I_\top(v, sk) = i_\top$$

$$I_\neg : (\mathbf{VAL\_QBF} \to \mathbf{BOOL}) \times \mathbf{VAL\_QBF} \to \mathbf{BOOL}$$

$$I_\neg(f)(v, sk) = i_\neg(f(v, sk))$$

$$I_\circ : (\mathbf{VAL\_QBF} \to \mathbf{BOOL})^2 \times \mathbf{VAL\_QBF} \to \mathbf{BOOL}$$

$$I_\circ(f, g)(v, sk) = i_\circ(f(v, sk), g(v, sk))$$
pour tout connecteur $\circ \in \{\wedge, \vee, \to, \leftrightarrow\}$

$$I^x_\exists, I^x_\forall : (\mathbf{VAL\_QBF} \to \mathbf{BOOL}) \times \mathbf{VAL\_QBF} \to \mathbf{BOOL}$$

$$I^x_\exists(f)(v, sk) = i_\vee(f(v[x := \mathbf{vrai}], sk), f(v[x := \mathbf{faux}], sk))$$
$$I^x_\forall(f)(v, sk) = i_\wedge(\ f(v[x := \mathbf{vrai}], sk(\mathbf{vrai})), f(v[x := \mathbf{faux}], sk(\mathbf{faux})))$$

Nous sommes en mesure de définir la sémantique des QBF comme une extension de la sémantique des connecteurs, constantes et quantificateurs.

Définition 3 (sémantique des QBF prénexes) *La sémantique d'une QBF prénexe F est une fonction*

$$I^*(F) : \mathbf{VAL\_QBF} \to \mathbf{BOOL}$$

définie inductivement par :
- $I^*(\bot)(v, sk) = I_\bot(v, sk)$;
- $I^*(\top)(v, sk) = I_\top(v, sk)$;
- $I^*(x)(v, sk) = v(x)$ *si* $x \in \mathcal{SP}$;
- $I^*((G \circ H))(v, sk) = I_\circ(I^*(G), I^*(H))(v, sk)$ *si* G, H *sont des QBF et* $\circ \in \{\wedge, \vee, \to, \leftrightarrow\}$;
- $I^*(\neg G)(v, sk) = I_\neg(I^*(G))(v, sk)$ *si* G *est une QBF.*
- $I^*((\exists x\ G))(v, sk) = I^*(G)(v[x := \hat{x}], sk \setminus \{(x \mapsto \hat{x})\})$ *si* G *est une QBF,* $x \in \mathcal{SP}$ *et* $(x \mapsto \hat{x}) \in sk$;
- $I^*((\exists x\ G))(v, sk) = I^x_\exists(I^*(G))(v, sk)$ *si* G *est une QBF,* $x \in \mathcal{SP}$ *et il n'existe pas de couple* $(x \mapsto \hat{x}) \in sk$ *(*$\hat{x}$ *fonction booléenne) ;*
- $I^*((\forall x\ G))(v, sk) = I^x_\forall(I^*(G))(v, sk)$ *si* G *est une QBF et* $x \in \mathcal{SP}$.

Cette sémantique est particulièrement souple puisqu'elle fait cohabiter le symbole propositionnel libre dont la valeur de vérité est déterminée par la valuation propositionnelle, le symbole propositionnel existentiellement quantifié associé à une fonction booléenne dont la valeur de vérité est déterminée lorsque, dans le processus de calcul inductif de la sémantique, le symbole propositionnel devient libre et enfin le symbole propositionnel existentiellement quantifié qui n'est pas associé à une fonction booléenne et qui s'élimine comme une disjonction sur les deux valeurs de vérité, retrouvant ainsi la sémantique du problème de décision. A notre connaissance, il n'existe pas de définition inductive de la sémantique des QBF prénexes qui permette une telle souplesse (la définition de [8] porte sur les QBF CNF et ne propose pas l'équivalent des valuations QBF partielles ; la définition de [11], si elle est inductive, ne manipule ni les symboles propositionnels libres, ni les valuations QBF partielles sauf pour décrire des solutions partielles à des QBF non valides). Cette sémantique met particulièrement aussi en lumière que restreinte à des formules sans quantificateur, elle n'est autre que la sémantique de la logique propositionnelle. Cette sémantique permet de retrouver aussi un résultat du premier ordre qui exprime que si la formule est close alors la valuation propositionnelle est sans importance.

Exemple 2 *En continuant l'exemple 1 et pour la valuation QBF partielle* $sk = \{(d \mapsto \hat{d})\}$ *(et une valuation propositionnelle v quelconque) avec*

$$\hat{d} = \{((\mathbf{vrai}, \mathbf{vrai}) \mapsto \mathbf{vrai}), ((\mathbf{vrai}, \mathbf{faux}) \mapsto \mathbf{vrai}), ((\mathbf{faux}, \mathbf{vrai}) \mapsto \mathbf{faux}), ((\mathbf{faux}, \mathbf{faux}) \mapsto \mathbf{faux})\}$$

$$\begin{aligned} & I^*(\forall a \exists b \forall c \exists d \mu)(v, sk) \\ = \ & I^a_\forall(I^*(\exists b \forall c \exists d \mu))(v, sk) \\ = \ & i_\wedge(\ I^*(\exists b \forall c \exists d \mu)(v[a := \mathbf{vrai}], sk(\mathbf{vrai})), \\ & \quad I^*(\exists b \forall c \exists d \mu)(v[a := \mathbf{faux}], sk(\mathbf{faux}))) \end{aligned}$$

Détaillons, avec $v_a = v[a := \mathbf{vrai}]$

$$\begin{aligned} & I^*(\exists b \forall c \exists d \mu)(v_a, sk(\mathbf{vrai})) \\ = \ & I^b_\exists(I^*(\forall c \exists d \mu))(v_a, sk(\mathbf{vrai})) \\ = \ & i_\vee(\ I^*(\forall c \exists d \mu)(v_a[b := \mathbf{vrai}], sk(\mathbf{vrai})), \\ & \quad I^*(\forall c \exists d \mu)(v_a[b := \mathbf{faux}], sk(\mathbf{vrai}))) \end{aligned}$$

avec $v_b = v_a[b := \mathbf{vrai}]$ *et* $sk(\mathbf{vrai}) = \{(d \mapsto \{(\mathbf{vrai} \mapsto \mathbf{vrai}), (\mathbf{faux} \mapsto \mathbf{vrai})\})\}$

$$\begin{aligned} & I^*(\forall c \exists d \mu)(v_b, \{(d \mapsto \hat{d}(\mathbf{vrai}))\}) = \\ & i_\wedge(\ I^*(\exists d \mu)(v_b[c := \mathbf{vrai}], \{(d \mapsto \hat{d}(\mathbf{vrai})(\mathbf{vrai}))\}), \\ & \quad I^*(\exists d \mu)(v_b[c := \mathbf{faux}], \{(d \mapsto \hat{d}(\mathbf{vrai})(\mathbf{faux}))\})) \end{aligned}$$

enfin en posant $v_c = v_b[d := \mathbf{vrai}]$

$$\begin{array}{ll} & I^*(\exists d\mu)(v_c, \{(d \mapsto \hat{d}(\mathbf{vrai})(\mathbf{vrai}))\}) \\ = & I^*(\mu)(v_c[d := \hat{d}(\mathbf{vrai})(\mathbf{vrai})], \emptyset) \\ = & I^*(\mu)(v_c[d := \mathbf{vrai}], \emptyset) \\ = & \mathbf{vrai} \end{array}$$

Sans plus de détails,

$$I^*(\forall a \exists b \forall c \exists d \mu)(v, sk) = \mathbf{vrai}.$$

La notion de modèle (propositionnel) est étendue aux QBF prénexes.

Définition 4 (modèle QBF) *Une valuation QBF V constituée d'une valuation propositionnelle v et d'une valuation fonctionnelle (totale) sk pour une QBF prénexe F est un modèle QBF pour F si $I^*(F)(v, sk) = \mathbf{vrai}$.*

Cette définition de la notion de modèle QBF n'est autre pour notre description de la sémantique que celle pour les QBF CNF introduite dans [8]. Lorsque la QBF prénexe est close alors la valuation propositionnelle n'importe pas et, dans la définition précédente, seule la valuation fonctionnelle participe au modèle de la QBF.

Exemple 3 *En continuant l'exemple 2, la valuation fonctionnelle $sk = \{(b \mapsto \hat{b}), (d \mapsto \hat{d})\}$ avec $\hat{b} = \{(\mathbf{vrai} \mapsto \mathbf{vrai}), (\mathbf{faux} \mapsto \mathbf{vrai})\}$ permet quelle que soit la valuation propositionnelle associée d'obtenir un modèle QBF pour la QBF prénexe $\forall a \exists b \forall c \exists d \mu$.*

Les modèles d'une formule propositionnelle et les modèles d'une QBF close prénexe quantifiée uniquement existentiellement dont la matrice est précisément la formule propositionnelle sont en bijection.

La sémantique attendue des quantificateurs peut être facilement retrouvée (G une QBF prénexe dont l'unique symbole propositionnel libre est x et une valuation propositionnelle v) :

$$\begin{array}{l} I^*((\forall x\ G))(v, \emptyset) = \\ \quad i_\wedge(I^*([x \leftarrow \top](G))(v, \emptyset), I^*([x \leftarrow \bot](G))(v, \emptyset)) \end{array}$$

et

$$\begin{array}{l} I^*((\exists x\ G))(v, \emptyset) = \\ \quad i_\vee(I^*([x \leftarrow \top](G))(v, \emptyset), I^*([x \leftarrow \bot](G))(v, \emptyset)). \end{array}$$

La définition classique de la validité d'une QBF close peut alors être redéfinie dans notre reformulation de la sémantique : une QBF close F est valide si, pour toute valuation propositionnelle v, $I^*(F)(v, \emptyset) = \mathbf{vrai}$.

La question du problème de validité peut être reformulée ainsi : une QBF close est valide si et seulement si elle admet un modèle QBF.

Le lemme suivant met en exergue le lien entre les modèles d'une QBF et la formule propositionnelle tautologique née de la substitution dans la matrice des symboles existentiellement quantifiés par des formules uniquement constituées des symboles universellement quantifiés.

Lemme 1 *Soit QF une QBF prénexe dont les variables existentielles sont $\{x_1, \ldots, x_n\}$, $sk = \{(x_i \mapsto \hat{x_i})\}$ un valuation fonctionnelle pour la QBF QF, $\phi_{\hat{x_1}}, \ldots, \phi_{\hat{x_n}}$ les formules associées positivement aux fonctions $\hat{x_1}, \ldots, \hat{x_n}$ et la substitution $\sigma = [x_1 \leftarrow \phi_{\hat{x_1}}] \ldots [x_n \leftarrow \phi_{\hat{x_n}}]$ alors la valuation QBF $V = (v, sk)$, v une valuation quelconque, est un modèle de la QBF QF si et seulement si la formule propositionnelle $\sigma(F)$ est une tautologie.*

Exemple 4 *En continuant l'exemple 3 et en considérant les formules propositionnelles $\phi_b = \mathbf{vrai}$ et $\phi_d = a$, associées aux fonctions booléennes $\hat{b}$ et $\hat{d}$, la formule propositionnelle $[b \leftarrow \phi_b][d \leftarrow \phi_d](\mu)$ est une tautologie.*

La sémantique des QBF prénexes suggère deux *relations d'équivalence* : la relation d'équivalence classique ($\equiv$) sur la préservation de la validité et la relation d'équivalence ($\cong$) sur la préservation des modèles [25]. La première s'exprime simplement dans notre sémantique (F et G deux QBF prénexes) : $F \equiv G$ si, pour toute valuation propositionnelle v, $I^*(F)(v, \emptyset) = I^*(G)(v, \emptyset)$.

La relation $\equiv$ ne préserve que la validité des QBF prénexes mais n'informe pas sur la préservation des modèles, ce qui est crucial pour par exemple la compilation des QBF ; nous avons donc proposé une nouvelle relation d'équivalence (dénotée $\cong$) sur la préservation des modèles [25].

Définition 5 (relation $\cong$) *Soient F et G deux QBF prénexes de lieurs identiques jusqu'à la dernière existentielle. $F \cong G$ si, pour toute valuation propositionnelle v et pour toute valuation fonctionnelle sk pour F et G, $I^*(F)(v, sk) = I^*(G)(v, sk)$.*

La définition proposée ici est différente de celle présentée initialement dans [25] car elle autorise à réunir dans une même classe d'équivalence des QBF prénexes ayant des lieurs qui diffèrent sur les universelles les plus internes, prenant ainsi en compte le fait qu'un modèle QBF est constitué de fonctions booléennes dépendant des symboles propositionnels universellement quantifiés qui précèdent le symbole propositionnel existentiellement quantifié associé à la fonction. Ainsi, bien que les lieurs soient différents $\exists x x \cong \exists x \forall y((x \vee y) \wedge (y \rightarrow x))$ car ces deux QBF prénexes ont pour modèles QBF $\{(x \mapsto \mathbf{vrai})\}$; mais

$\exists xx \not\cong \forall y \exists x((x \vee y) \wedge (y \rightarrow x))$ car l'unique modèle QBF de cette dernière QBF prénexe est la valuation QBF $(\emptyset, \{(x \mapsto \{(\mathbf{vrai} \mapsto \mathbf{vrai}), (\mathbf{faux} \mapsto \mathbf{vrai})\})\})$. Une autre définition, plus complexe, de la relation d'équivalence préservant les modèles peut être choisie de telle manière qu'elle ne tienne pas compte de l'ordre des quantificateurs universels contigus.

La différence entre les deux relations d'équivalence est la suivante : la relation d'équivalence préservant uniquement la validité autorise les valuations QBF partielles ; ainsi $\exists xx \equiv \top$ mais $\exists xx \not\cong \top$ car l'unique modèle QBF de $\top$ est la valuation QBF vide alors que l'unique modèle QBF de $\exists xx$ est la valuation QBF $\{(x \mapsto \mathbf{vrai})\}$. De même $\exists xx \equiv \exists yy$ mais $\exists xx \not\cong \exists yy$.

Le lemme suivant établit que la relation $\cong$ est bien une relation d'équivalence.

Lemme 2 *La relation $\cong$ est une relation d'équivalence pour les QBF prénexes.*

Ce lemme est immédiat car l'égalité est une relation d'équivalence.

La relation d'équivalence préservant les modèles réalise une partition de la classe d'équivalence de représentant $\top$ de la relation d'équivalence préservant la validité : c'est ce qu'exprime le premier item du lemme suivant ; le second item exprime que les deux relations sont identiques lorsque les QBF sont sans quantificateur. La preuve du lemme est immédiate à partir de la remarque ci-dessus et des définitions.

Lemme 3 *Soient F et G deux QBF prénexes.*
- *Si $F \cong G$ alors $F \equiv G$.*
- *De plus si F et G sont deux QBF sans quantificateur alors $F \equiv G$ si et seulement si $F \cong G$.*

Le lemme suivant établit la congruence de l'équivalence de préservation des modèles par rapport aux quantifications existentielle et universelle et les corollaires pour des QBF sans quantificateur.

Lemme 4 *Soient F et G deux QBF prénexes. Si $F \cong G$ et x un symbole propositionnel lié ni dans F ni dans G alors $\exists xF \cong \exists xG$ et $\forall xF \cong \forall xG$.*

De plus, si F et G sont sans quantificateur, $F \equiv G$ et x un symbole propositionnel alors $\exists xF \cong \exists xG$ et $\forall xF \cong \forall xG$.

Les résultats classiques s'étendent partiellement à la relation d'équivalence préservant les modèles.

Lemme 5 *Soient F et G deux QBF prénexes, H une formule propositionnelle, x et y deux symboles propositionnels et Q un lieur. Alors*

($Q^{\cong}$.1) $\exists x \exists y F \cong \exists y \exists x F$

($Q^{\cong}$.2) $I^*(\forall x \forall y F)(v, sk) = I^*(\forall y \forall x F)(v, sk')$ *avec sk et sk' identiques à ceci près que les deux premières colonnes sont permutées*

($Q^{\cong}$.3) $\exists x Q(x \wedge H) \cong \exists x Q(x \wedge [x \leftarrow \top](H))$

($Q^{\cong}$.4) $\exists x Q(\neg x \wedge H) \cong \exists x Q(\neg x \wedge [x \leftarrow \bot](H))$

($Q^{\cong}$.5) $I^*(\forall x Q(x \vee H))(v, sk) = I^*(Q[x \leftarrow \bot](H))(v, sk(\mathbf{faux}))$

($Q^{\cong}$.6) $I^*(\forall x Q(\neg x \vee H))(v, sk) = I^*(Q[x \leftarrow \top](H))(v, sk(\mathbf{vrai}))$

Les équivalences $Q^{\cong}.3$, $Q^{\cong}.4$, $Q^{\cong}.5$ et $Q^{\cong}.6$ réalisent une propagation unitaire : les équivalences $Q^{\cong}.3$ et $Q^{\cong}.4$ conservent le symbole propositionnel existentiellement quantifié dans la matrice pour préserver les modèles (puisque le lieur doit être le même de part et d'autre) tandis que $Q^{\cong}.5$ et $Q^{\cong}.6$ éliminent le symbole propositionnel universellement quantifié ainsi que son quantificateur (les QBF ne pouvant alors plus être équivalentes). L'exemple suivant démontre par un exemple que $\forall x \forall y F \not\cong \forall y \forall x F$ ce qui met en exergue une disymétrie par l'équivalence $Q^{\cong}.1$ dans le comportement des deux quantificateurs.

Exemple 5 *Soient les deux QBF prénexes $\theta = \forall a \forall b \exists c((\neg a \wedge b) \rightarrow c)$ et $\theta' = \forall b \forall a \exists c((\neg a \wedge b) \rightarrow c)$ ainsi que la valuation fonctionnelle*

$$sk = \{(c \mapsto \{((\mathbf{vrai}, \mathbf{vrai}) \mapsto \mathbf{vrai}), ((\mathbf{vrai}, \mathbf{faux}) \mapsto \mathbf{vrai}), ((\mathbf{faux}, \mathbf{vrai}) \mapsto \mathbf{faux}), ((\mathbf{faux}, \mathbf{faux}) \mapsto \mathbf{faux})\})\}$$

alors

$$I^*(\theta)(v, sk) = \mathbf{faux} \neq I^*(\theta')(v, sk) = \mathbf{vrai}$$

mais avec la valuation fonctionnelle suivante qui respecte vis-à-vis de sk les conditions de $Q^{\cong}.2$

$$sk' = \{(c \mapsto \{((\mathbf{vrai}, \mathbf{vrai}) \mapsto \mathbf{vrai}), ((\mathbf{faux}, \mathbf{vrai}) \mapsto \mathbf{vrai}), ((\mathbf{vrai}, \mathbf{faux}) \mapsto \mathbf{faux}), ((\mathbf{faux}, \mathbf{faux}) \mapsto \mathbf{faux})\})\}$$

alors $I^*(\theta)(v, sk) = I^*(\theta')(v, sk')$.

4 Calcul syntaxique

Nous présentons notre calcul syntaxique comme une extension pour les QBF de la justification dans le cadre de la théorie de la preuve de l'algorithme de Stålmarck [24].

4.1 Relation de formules

Une relation de formules [24] (propositionnelle) R pour une formule F est définie comme étant une relation d'équivalence sur $sub(F)$ avec pour contrainte

que pour tout éléments $A, B \in sub(F)$, si $R(A, B)$ alors $R(\overline{A}, \overline{B})$. Une classe d'une relation de formules R contenant un élément A est notée $[A]_R$ (et plus simplement $[A]$ lorsque la relation de formules est clairement explicitée par le contexte). La sémantique attendue est que tous les éléments d'une même classe d'équivalence ont la même valeur de vérité. Nous étendons ce formalisme aux QBF QM en ajoutant un lieur à la partition P de $sub(M)$ et en écrivant la relation de formules (Q, P).

La relation de formules la plus fine pour une QBF prénexe QM est la relation identité Id_{QM} qui est la partition des singletons de $sub(M)$ associée au lieur Q. La relation de formules $R([A]_R = [B]_R)$ (notée plus simplement par la suite $R([A] = [B])$) pour une QBF prénexe QM est la relation de formules la plus fine pour QM telle qu'elle est un raffinement de R et $([A]_R = [B]_R)$. La relation de formules $Id_{QM}([M] = [\top])$ nous servira de racine à l'arbre de déduction de notre calcul syntaxique. Dans ce qui suit seule une des deux classes symétriques $[A]$ et $[\overline{A}]$ sera explicitée.

Exemple 6 *Soit la QBF*

$$\xi = \forall a \exists b \forall c \exists d \neg((c \rightarrow b) \rightarrow \neg(d \rightarrow a))$$

alors

$$Id_\xi([\overline{F}] = [\top]) = (\forall a \exists b \forall c \exists d, [\top, \overline{F}], [a], [b], [c], [d], [F_0], [F_1])$$

avec $F = (F_0 \rightarrow \overline{F_1})$, $F_0 = (c \rightarrow b)$ *et* $F_1 = (d \rightarrow a)$.

4.2 Système syntaxique

Nous présentons notre système syntaxique S_{QBF} pour les QBF comme un ensemble de règles $\frac{conclusion}{prémisse}$ qui opèrent sur des relations de formules formant un arbre de déduction dont la racine est la relation de formules $Id_{QM}([M] = [\top])$. Mais avant de présenter le système nous définissons un ensemble de relations de formules qui correspondent à des relations de formules à partir desquelles aucune déduction ne sera plus possible.

Définition 6 (explicitement contradictoire) *Une relation de formules est* explicitement contradictoire *si une des conditions suivantes est vérifiée :*

1. *il existe une formule F dans la relation de formules telle que $[F] = [\overline{F}]$;*
2. *il existe une classe qui contient au moins deux symboles propositionnels universellement quantifiés du lieur ;*
3. *il existe une classe qui contient un symbole propositionnel universellement quantifié u du lieur et un symbole propositionnel existentiellement quantifié e tel que $e < u$.*

La condition 1 maintient intuitivement qu'une formule et son complémentaire ne peuvent avoir la même valeur de vérité. La condition 2 exprime que deux symboles universellement quantifiés ne sont jamais liés fonctionnellement. La condition 3 correspond dans le cadre de l'unification de termes du premier ordre au classique test d'occurrence. Une relation de fomules peut n'être pas explicitement contradictoire et contenir des classes qui le sont.

Nous ne présentons les règles du système S_{QBF} que pour le fragment $\{\rightarrow, \top\}$. Pour simplifier, dans la description de la prémisse et de la conclusion de la règle, seuls le lieur et les classes pertinentes sont notées, les classes invariantes ne sont pas décrites (la classe $[\top]$ est implicitement toujours présente).

Définition 7 (système S_{QBF} pour $\{\rightarrow, \top\}$) *Le système S_{QBF} est constitué de deux règles d'élimination du quantificateur existentiel :*

$$\exists\top : \frac{(Q,[x]=[\top])}{(\exists x Q,[x])} \qquad \exists\bot : \frac{(Q,[\overline{x}]=[\top])}{(\exists x Q,[x])}$$

d'une règle d'élimination du quantificateur universel :

$$\forall : \frac{(Q,[x]=[\top]) \quad (Q,[\overline{x}]=[\top])}{(\forall x Q,[x])}$$

et neuf règles de fusion des classes :

$$1 : \frac{(Q,[\overline{F_X}]=[\top],[\overline{F_Y}]=[\top])}{(Q,[(F_X \rightarrow F_Y)]=[\overline{F_Y}])} \qquad 2 : \frac{(Q,[F_X]=[\top],[F_Y]=[\top])}{(Q,[(F_X \rightarrow F_Y)]=[\overline{F_X}])}$$

$$3 : \frac{(Q,[F_X]=[\top],[\overline{F_Y}]=[\top])}{(Q,[(F_X \rightarrow F_Y)]=[\top])} \qquad 4 : \frac{(Q,[(F_X \rightarrow F_Y)]=[\overline{F_X}])}{(Q,[(F_X \rightarrow F_Y)],[\overline{F_Y}]=[\top])}$$

$$5 : \frac{(Q,[(F_X \rightarrow F_Y)]=[\overline{F_X}])}{(Q,[(F_X \rightarrow F_Y)],[F_X]=[\overline{F_Y}])} \qquad 6 : \frac{(Q,[(F_X \rightarrow F_Y)]=[\top])}{(Q,[(F_X \rightarrow F_Y)],[\overline{F_X}]=[\top])}$$

$$7 : \frac{(Q,[(F_X \rightarrow F_Y)]=[\top])}{(Q,[(F_X \rightarrow F_Y)],[F_Y]=[\top])} \qquad 8 : \frac{(Q,[(F_X \rightarrow F_Y)]=[\top])}{(Q,[(F_X \rightarrow F_Y)],[F_X]=[F_Y])}$$

$$9 : \frac{(Q,[(F_X \rightarrow F_Y)]=[F_Y])}{(Q,[(F_X \rightarrow F_Y)],[F_X]=[\top])}$$

Le lien sémantique entre la prémisse et la conclusion des règles de fusion est un lien d'équivalence de préservation des modèles qui sera explicité dans la section suivante.

Exemple 7 *La règle 9 de la définition précédente exprime que si la relation de formules R est telle que son lieur est Q et la partition est telle qu'une classe contienne une formule $(F_X \rightarrow F_Y)$ et que les formules F_X et $\top$ appartiennent à la même classe alors est inférée par la règle 9 une relation de formules $R([(F_X \rightarrow F_Y)] = [F_Y])$ (en particulier, cette nouvelle relation de formules est toujours telle que $[F_X] = [\top]$).*

Nous sommes à même de décrire ce qu'est un arbre de déduction et une preuve pour le système S_{QBF}.

Définition 8 (arbre de déduction) *Un* arbre de déduction *est défini inductivement ainsi :*

- *toute relation de formules qui est non explicitement contradictoire est un arbre de déduction pour le système S_{QBF} ;*
- *si R est une relation de formules, r une règle du S_{QBF} telle que la premisse soit satisfaite par R et que le résultat R', unique conclusion de l'application de la règle à la prémisse soit non explicitement contradictoire et si ∇' est un arbre de déduction de racine R' alors $\frac{\nabla'}{R}r$ est un arbre de déduction de racine R ;*
- *si R est une relation de formules, r une règle du S_{QBF} telle que la premisse soit satisfaite par R et que les résultats R' et R'', conclusions de l'application de la règle r à la prémisse soient non explicitement contradictoires et si ∇' et ∇'' sont des arbres de déduction de racine respectivement R' et R'' alors $\frac{\nabla' \quad \nabla''}{R}r$ est un arbre de déduction de racine R.*

Dans les exemples qui suivent, seule l'application des règles de fusion est indicée par le numéro de la règle.

Exemple 8 (suite de l'exemple 6) *L'arbre ci-dessous est un arbre de déduction pour le système S_{QBF} ($F = (F_0 \to \overline{F_1})$, $F_0 = (c \to b)$ et $F_1 = (d \to a)$).*

$$\dfrac{\dfrac{\nabla \quad \nabla'}{(\forall a \exists b \forall c \exists d, [\top, \overline{F}, F_0, F_1], [a], [b], [c], [d])}\,3}{Id_\xi([\overline{F}] = [\top])}$$

avec ∇ tel que

$$\dfrac{\dfrac{\dfrac{(\varepsilon, [\top, \overline{F}, F_0, F_1, a, b, c, d])}{(\exists d, [\top, \overline{F}, F_0, F_1, a, b, c], [d])} \quad \dfrac{(\varepsilon, [\top, \overline{F}, F_0, F_1, a, b, \overline{c}, d])}{(\exists d, [\top, \overline{F}, F_0, F_1, a, b, \overline{c}], [d])}}{(\forall c \exists d, [\top, \overline{F}, F_0, F_1, a, b], [c], [d])}}{(\exists b \forall c \exists d, [\top, \overline{F}, F_0, F_1, a], [b], [c], [d])}$$

avec ∇' tel que

$$\dfrac{\dfrac{\dfrac{(\exists d, [\top, \overline{F}, F_0, F_1, \overline{a}, \overline{d}, b, c]) \quad (\exists d, [\top, \overline{F}, F_0, F_1, \overline{a}, \overline{d}, b, \overline{c}])}{(\forall c \exists d, [\top, \overline{F}, F_0, F_1, \overline{a}, \overline{d}, b], [c])}}{(\exists b \forall c \exists d, [\top, \overline{F}, F_0, F_1, \overline{a}, \overline{d}], [b], [c])}}{(\exists b \forall c \exists d, [\top, \overline{F}, F_0, F_1, \overline{a}], [b], [c], [d])}\,4$$

La relation de formules $(\exists d, [\top, \overline{F}, F_0, F_1, \overline{a}, \overline{d}, b, c])$ n'est pas explicitement contradictoire bien que a et c soient des symboles propositionnels universellement quantifés car ils n'appartiennent plus au lieur qui est présentement $\exists d$.

Nous sommes en mesure de définir ce qu'est une preuve pour une QBF dans le système S_{QBF}.

Définition 9 (axiome et preuve) *Un* axiome *est une relation de formules non explicitement contradictoire ne contenant que deux classes et qui est telle que l'on ne puisse construire un arbre de déduction avec l'axiome pour racine et contenant une relation de formules explicitement contradictoire. Une* preuve *pour une QBF QM dans le système S_{QBF} est un arbre de déduction de racine $Id_{QM}([M] = [\top])$ tel que toute feuille soit un axiome.*

Exemple 9 *L'arbre de déduction de l'exemple 8 est une preuve dans le système S_{QBF} de la QBF*

$$\forall a \exists b \forall c \exists d \neg((c \to b) \to \neg(d \to a)).$$

La définition de l'axiome prévient la contradiction dans une classe. Cette définition se réduit uniquement à la première condition si l'utilisation de la règle d'élimination du quantificateur n'est utilisée que lorsque aucune autre règle ne peut plus l'être (car alors nécessairement des règles aurait été déduite la contradiction).

Exemple 10 *La relation de formules*

$$(\varepsilon, [(a \to b), a, \overline{b}, \top], [\overline{(a \to b)}, \overline{a}, b, \overline{\top}])$$

(dont exceptionnellement toutes les classes ont été explicitées) n'est pas explicitement contradictoire mais n'est pas un axiome car :

$$\dfrac{(\varepsilon, [(a \to b), a, \overline{b}, \top, \overline{(a \to b)}, \overline{a}, b, \overline{\top}])}{(\varepsilon, [(a \to b), a, \overline{b}, \top], [\overline{(a \to b)}, \overline{a}, b, \overline{\top}])}\,1$$

Ceci interdit une preuve n'utilisant que la règle d'élimination du quantificateur existentiel telle que :

$$\dfrac{\dfrac{(\varepsilon, [(a \to b), a, \overline{b}, \top], [\overline{(a \to b)}, \overline{a}, b, \overline{\top}])}{(\exists b, [(a \to b), a, \top], [\overline{(a \to b)}, \overline{a}, \overline{\top}], [b], [\overline{b}])}}{(\exists a \exists b, [(a \to b), \top], [\overline{(a \to b)}, \overline{\top}], [a], [\overline{a}], [b], [\overline{b}])}$$

4.3 Sémantique

Nous introduisons une fonction d'interprétation qui explicite la sémantique d'une relation de fonctions en une QBF.

Définition 10 (fonction d'interprétation) *La fonction d'interprétation d'une relation de formules $(\cdot, \cdot)^*$ est une fonction de l'ensemble des relations de formules dans les QBF définie par*

$$(Q, P)^* = Q \bigwedge_{C \in P} \Big(\bigwedge_{F, F' \in C} (F \leftrightarrow F') \Big).$$

Clairement, $(Q, Id_{QM}([M] = [\top]))^* \cong QM$.

Exemple 11

$$(\exists b \forall c \exists d, [\top, \overline{F}, F_0, F_1, \overline{a}, \overline{d}], [b], [c])^* \\ = \quad \exists b \forall c \exists d (\overline{F} \wedge F_0 \wedge F_1 \wedge \neg a \wedge \neg d)$$

La correction du système S_{QBF} par rapport à la sémantique des QBF est une conséquence du lemme suivant.

Lemme 6 (Correction des règles de fusion) *Soit la relation de formules R' le résultat de l'application d'une règle de fusion sur une relation de formules R alors $R'^* \cong R^*$.*

Il est immédiat qu'à partir du calcul de la validité directement par la définition de la sémantique des quantificateurs il est possible de construire une preuve dans le système S_{QBF} : la preuve, de bas en haut élimine tous les quantifcateurs puis fusionne les classes grâce aux règles 4, 5 et 7, d'où la complétude.

Théorème 1 (Correction et complétude du système S_{QBF} par rapport à la sémantique des QBF) *Une QBF QM est valide si et seulement si la relation de formules $Id_{QM}([M] = [\top])$ admet une preuve dans le système S_{QBF}.*

4.4 Extensions au système S_{QBF}

Le système S_{QBF} présenté jusqu'ici est assez pauvre : il n'élimine les quantificateurs qu'en employant explicitement leur sémantique et il ne tire aucun avantage des propriétés algébriques du connecteur logique. Les extensions proposées ci-dessous préservent la correction (la complétude étant préservée si nous ne faisons qu'étendre le système de règles).

Nouvelles règles d'élimination triviale des quantificateurs. Nous introduisons deux règles d'élimination des quantificateurs pour les symboles propositionnels qui n'interviennent pas dans la matrice d'une QBF :

$$\not\exists : \ \frac{(QQ', P)}{(Q \exists x Q', P)} \qquad \not\forall : \ \frac{(QQ', P)}{(Q \forall x Q', P)}$$

sachant que le symbole propositionnel x n'apparait pas dans la partition P.

Nouvelles règles pour l'élimination des quantificateurs. Nous introduisons deux nouvelles règles d'élimination du quantificateur existentiel basées sur l'équivalence : $Q \exists x Q'(x \wedge H) \equiv QQ'[x \leftarrow \top](H)$.

$$\exists + : \ \frac{(QQ', [x]=[\top])}{(Q \exists x Q', [x]=[\top])} \qquad \exists - : \ \frac{(QQ', [\overline{x}]=[\top])}{(Q \exists x Q', [\overline{x}]=[\top])}$$

Elles expriment que si le symbole propositionnel existentiellement quantifié a été déduit comme ne pouvant valoir qu'une valeur de vérité particulière alors le quantificateur peut être supprimé.

Nouvelles règles pour les symboles propositionnels universellement quantifiés. L'ensemble des règles de fusion sont purement propositionnelles et ne tiennent pas compte des contraintes qui s'imposent pour les symboles propositionnels universellement quantifiés. Nous introduisons un nouvel ensemble de règles de fusion qui se déduisent aisément de la sémantique associée à une relation de formules via la fonction d'interprétation.

$$10 : \ \frac{(QQ' \forall x Q'', [y]=[\top])}{(Q \exists y Q' \forall x Q'', [(x \rightarrow y)]=[\top])}$$

$$11 : \ \frac{(QQ' \forall x Q'', [y]=[\top])}{(Q \exists y Q' \forall x Q'', [(x \rightarrow y)]=[y])}$$

$$12 : \ \frac{(QQ' \forall y Q'', [x]=[\top])}{(Q \exists x Q' \forall y Q'', [(x \rightarrow y)]=[y])}$$

$$13 : \ \frac{(QQ' \forall x Q'', [\overline{y}]=[\top])}{(Q \exists y Q' \forall x Q'', [(x \rightarrow y)]=[\overline{x}])}$$

$$14 : \ \frac{(QQ' \forall y Q'', [\overline{x}]=[\top])}{(Q \exists x Q' \forall y Q'', [(x \rightarrow y)]=[\top])}$$

$$15 : \ \frac{(QQ' \forall y Q'', [\overline{x}]=[\top])}{(Q \exists x Q' \forall y Q'', [(x \rightarrow y)]=[\overline{x}])}$$

Le nouveau système peut être considéré comme l'abstraction d'une restriction du système de propagation de contraintes pour les QBF [26] en voyant la propagation de contraintes comme un système de fusion des classes d'équivalences.

Exemple 12 *En continuant l'exemple 1. Extrait de la preuve pour le système S_{QBF} enrichi (avec $F_0 = (c \rightarrow b)$) :*

$$\dfrac{\dfrac{(\forall a \forall c \exists d, [\top, \overline{F}, F_0, F_1, b], [a], [c], [d])}{(\forall a \exists b \forall c \exists d, [\top, \overline{F}, F_0, F_1], [a], [b], [c], [d])}\,{}_{10}}{Id_\xi([\overline{F}] = [\top])}\,{}_{3}$$

4.5 Calcul du modèle

Nous annotons notre système S_{QBF} pour les QBF prénexes en y ajoutant la construction d'un modèle QBF : nos règles opérent maintenant sur un couple $V \vdash (Q, P)$ avec P une partition pour $sub(M)$, (Q, P) une relation de formules et V une valuation QBF de QM. Nous étendons les notions d'axiome, de dérivation et de preuve pour un nouveau système S^a_{QBF}, extension annotée du système S_{QBF}.

Le système S^a_{QBF} est constitué de deux règles d'élimination du quantificateur existentiel :

$$\exists \top^a : \ \frac{(v[x{:=}\mathbf{vrai}], sk) \vdash (Q, [x]=[\top])}{(v, \{\hat{x}{=}\mathbf{vrai}\} \cup sk) \vdash (\exists x Q, [x])}$$

$$\exists \bot^a : \ \frac{(v[x{:=}\mathbf{faux}], sk) \vdash (Q, [\overline{x}]=[\top])}{(v, \{\hat{x}{=}\mathbf{faux}\} \cup sk) \vdash (\exists x Q, [x])}$$

d'une règle d'élimination du quantificateur universel (en notant $V_{\mathbf{vrai}} = (v[x := \mathbf{vrai}], sk(\mathbf{vrai}))$ et $V_{\mathbf{faux}} = (v[x := \mathbf{faux}], sk(\mathbf{faux}))$) :

$$\forall^a : \quad \frac{V_{\mathbf{vrai}} \vdash (Q,[x]=[\top]) \quad V_{\mathbf{faux}} \vdash (Q,[\overline{x}]=[\top])}{(v,sk) \vdash (\forall x Q,[x])}$$

et chaque règle de fusion des classes de 1 à 9 de la définition 7 de structure $\frac{R'}{R}$ est augmentée en une règle $\frac{V \vdash R'}{V \vdash R}$.

Nous annotons aussi les règles de fusion 10 à 15 de l'extension du système S_{QBF} :

$$10 : \quad \frac{(v[y:=\mathbf{vrai}],sk) \vdash (QQ'\forall xQ'',[y]=[\top])}{(v,\{\hat{y}=\mathbf{vrai}\}\cup sk) \vdash (Q\exists yQ'\forall xQ'',[(x\rightarrow y)]=[\top])}$$

$$11 : \quad \frac{(v[y:=\mathbf{vrai}],sk) \vdash (QQ'\forall xQ'',[y]=[\top])}{(v,\{\hat{y}=\mathbf{vrai}\}\cup sk) \vdash (Q\exists yQ'\forall xQ'',[(x\rightarrow y)]=[y])}$$

$$12 : \quad \frac{(v[x:=\mathbf{vrai}],sk) \vdash (QQ'\forall yQ'',[x]=[\top])}{(v,\{\hat{x}=\mathbf{vrai}\}\cup sk) \vdash (Q\exists xQ'\forall yQ'',[(x\rightarrow y)]=[y])}$$

$$13 : \quad \frac{(v[y:=\mathbf{faux}],sk) \vdash (Q\exists yQ'\forall xQ'',[\overline{y}]=[\top])}{(v,\{\hat{y}=\mathbf{faux}\}\cup sk) \vdash (Q\exists yQ'\forall xQ'',[(x\rightarrow y)]=[\overline{x}])}$$

$$14 : \quad \frac{(v[x:=\mathbf{faux}],sk) \vdash (QQ'\forall yQ'',[\overline{x}]=[\top])}{(v,\{\hat{x}=\mathbf{faux}\}\cup sk) \vdash (Q\exists xQ'\forall yQ'',[(x\rightarrow y)]=[\top])}$$

$$15 : \quad \frac{(v[x:=\mathbf{faux}],sk) \vdash (QQ'\forall yQ'',[\overline{x}]=[\top])}{(v,\{\hat{x}=\mathbf{faux}\}\cup sk) \vdash (Q\exists xQ'\forall yQ'',[(x\rightarrow y)]=[\overline{x}])}$$

Si la preuve se construit de bas en haut pour la recherche des axiomes, la construction du modèle QBF se réalise de haut en bas.

Exemple 13 *En continuant l'exemple 1 et pour la valuation QBF* $sk = \{(b \mapsto \hat{b}), (d \mapsto \hat{d})\}$ *(et une valuation propositionnelle* v *quelconque) avec*

$\hat{b} = \{(\mathbf{vrai} \mapsto \mathbf{vrai}), (\mathbf{faux} \mapsto \mathbf{vrai})\} = \mathbf{vrai}$ *et*
$\hat{d} = \{((\mathbf{vrai}, \mathbf{vrai}) \mapsto \mathbf{vrai}), ((\mathbf{vrai}, \mathbf{faux}) \mapsto \mathbf{vrai}),$
$((\mathbf{faux}, \mathbf{vrai}) \mapsto \mathbf{faux}), ((\mathbf{faux}, \mathbf{faux}) \mapsto \mathbf{faux})\}$

Extrait de la preuve pour le système S^a_{QBF} *avec calcul du modèle :*

$$\dfrac{\dfrac{\dfrac{\nabla_{\mathbf{vrai}} \quad \nabla_{\mathbf{faux}}}{(v[b := \mathbf{vrai}], \{(d \mapsto \hat{d})\}) \vdash (\forall a \forall c \exists d, [\top, \overline{F}, F_0, F_1, b], [a], [c], [d])}}{(v, sk) \vdash (\forall a \exists b \forall c \exists d, [\top, \overline{F}, F_0, F_1], [a], [b], [c], [d])}\,{}_{3}}{(v, sk) \vdash Id_\xi([\overline{F}] = [\top])}\,{}_{10}$$

avec $\nabla_{\mathbf{vrai}}$ *tel que*

$$(v[b := \mathbf{vrai}][a := \mathbf{vrai}], \{(d \mapsto \hat{d}(\mathbf{vrai}))\}) \vdash (\forall c \exists d, [\top, \overline{F}, F_0, F_1, b, a], [c], [d])$$

et $\nabla_{\mathbf{faux}}$ *tel que*

$$(v[b := \mathbf{vrai}][a := \mathbf{faux}], \{(d \mapsto \hat{d}(\mathbf{faux}))\}) \vdash (\forall c \exists d, [\top, \overline{F}, F_0, F_1, b, \overline{a}], [c], [d])$$

5 Conclusion

Nous avons proposé dans cet article une nouvelle présentation de la sémantique des QBF souple permettant de traiter les symboles propositionnels libres, les symboles propositionnels existentiellement quantifiés associés à une fonction booléenne dans une valuation QBF et les symboles propositionnels existentiellement quantifiés qui ne sont pas associés à une fonction booléenne dans une valuation QBF et qui sont éliminés suivant la sémantique habituelle basée sur la disjonction. Cette sémantique nous a permis d'explorer une relation d'équivalence basée sur la préservation des modèles QBF et non seulement sur la préservation de la validité. Nous avons proposé en outre un calcul syntaxique démontré correct et complet vis-à-vis de la sémantique des QBF qui tire parti de cette nouvelle relation d'équivalence en étendant la notion de relation de formules aux QBF. Le travail le plus proche de cette dernière contribution est sans doute le système de séquent GQBF [13] pour QBF CNF non prénexes sur lequel est basé la procédure `qpro`. Dans ce travail sont présentes les règles d'élimination des quantificateurs $\exists\top$, $\exists\bot$ et $\forall$ qui sont inhérentes à la sémantique des QBF. Ce travail est une extension du calcul des séquents classiques [14] orienté élimination des connecteurs et non relation de formules et son principal intérêt, de notre point de vue, est de fournir une justification dans le cadre de la théorie de preuve à la technique du «backjumping» [21, 15]. Notre proposition de calcul syntaxique non seulement étend aux QBF le système proposé pour l'algorithme de Stålmarck [24] mais l'étend aussi au niveau propositionnel dans la mesure où il permet de définir, dans le cadre de la propagation de contraintes, des propagateurs plus puissants (i.e. qui propagent plus d'information).

Références

[1] A. Ayari and D. Basin. Qubos : Deciding Quantified Boolean Logic using Propositional Satisfiability Solvers. In *Proceedings of the 4th International Conference on Formal Methods in Computer-Aided Design (FMCAD'02)*, pages 187–201, 2002.

[2] M. Benedetti. Evaluating QBFs via Symbolic Skolemization. In *Proceedings of the 11th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR'05)*, number 3452 in LNCS, pages 285–300. Springer, 2005.

[3] M. Benedetti. Extracting Certificates from Quantified Boolean Formulas. In *Proceedings of 9th International Joint Conference on Artificial Intelligence (IJCAI'05)*, pages 47–53, 2005.

[4] M. Benedetti and H. Mangassarian. Experience and Perspectives in QBF-Based Formal Verification. *Journal on Satisfiability, Boolean Modeling and Computation*, 5 :133–191, 2008.

[5] P. Besnard, T. Schaub, H. Tompits, and S. Woltran. Paraconsistent reasoning via quantified Boolean formulas, i : Axiomatising signed systems. In *In Proceedings of the 8th European Conference on Logics in Artificial Intelligence (JELIA'02)*, pages 320–331, 2002.

[6] A. Biere. Resolve and Expand. In *Proceedings of the 7th International Confrerence on Theory and Applications of Satisfiability Testing (SAT'04)*, pages 59–70, 2004.

[7] H. K. Büning, M. Karpinski, and A. Flögel. Resolution for quantified Boolean formulas. *Information and Computation*, 117(1) :12–18, 1995.

[8] H. K. Büning, K. Subramani, and X. Zhao. Boolean Functions as Models for Quantified Boolean Formulas. *Journal of Automated Reasoning*, 39(1) :49–75, 2007.

[9] M. Cadoli, A. Giovanardi, and M. Schaerf. Experimental Analysis of the Computational Cost of Evaluating Quantified Boolean Formulae. In *Fifth Conference of the Italian Association for Artificial Intelligence*, pages 207–218, 1997.

[10] M. Cadoli, M. Schaerf, A. Giovanardi, and M. Giovanardi. An Algorithm to Evaluate Quantified Boolean Formulae and Its Experimental Evaluation. *Journal of Automated Reasoning*, 28(2) :101–142, 2002.

[11] S. Coste-Marquis, H. Fargier, J. Lang, D. Le Berre, and P. Marquis. Representing Policies for Quantified Boolean Formulae. In *Proceedings of the 10th International Conference on Principles of Knowledge Representation and Reasoning (KR'06)*, pages 286–296, 2006.

[12] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Communication of the ACM*, 5, 1962.

[13] U. Egly, M. Seidl, and S. Woltran. A Solver for QBFs in Nonprenex Form. *Constraints*, 14(1) :38–79, 2009.

[14] J. H. Gallier. *Logic for computer science : foundations of automatic theorem proving*. Harper & Row Publishers, Inc., 1985.

[15] E. Giunchiglia, M. Narizzano, and A. Tacchella. Backjumping for Quantified Boolean Logic Satisfiability. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI'01)*, pages 275–281, 2001.

[16] E. Giunchiglia, M. Narizzano, and A. Tacchella. Clause/Term Resolution and Learning in the Evaluation of Quantified Boolean Formulas. *Journal of Artificial Intelligence Research*, 26 :371–416, 2006.

[17] F. Bacchus H. Samulowitz. Using SAT in QBF. In *Proceedings of the 11th International Conference on Principles and Practice of Constraint Programming, Lecture Notes in Computer Science*, volume 3709, pages 578 – 592, 2005.

[18] A.R. Meyer and L.J. Stockmeyer. The equivalence problem for regular expressions with squaring requires exponential space. In *Proceedings of the 13th Annual Symposium on Switching and Automata Theory (SWAT'72)*, pages 125–129, 1972.

[19] G. Pan and M.Y. Vardi. Symbolic Decision Procedures for QBF. In *International Conference on Principles and Practice of Constraint Programming*, 2004.

[20] D.A. Plaisted, A. Biere, and Y. Zhu. A satisfiability procedure for quantified Boolean formulae. *Discrete Applied Mathematics*, 130 :291–328, 2003.

[21] Patrick Prosser. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, 9 :268–299, 1993.

[22] J. Rintanen. Constructing conditional plans by a theorem-prover. *Journal of Artificial Intelligence Research*, 10 :323–352, 1999.

[23] J.A. Robinson. A machine-oriented logic based on the resolution principle. *JACM*, 12(1) :23–41, 1965.

[24] M. Sheeran and G. Stålmarck. A tutorial on Stålmarck's proof procedure for propositional logic. In G. Gopalakrishnan and P. Windley, editors, *Formal Methods in Computer-Aided Design*, volume 1522, pages 82–99. Springer-Verlag, Berlin, 1998.

[25] I. Stéphan. Algorithmes d'élimination de quantificateurs pour le calcul des politiques des formules booléennes quantifiées. In *Premières Journées Francophones de Programmation par Contraintes*, 2005.

[26] I. Stéphan. Boolean Propagation Based on Literals for Quantified Boolean Formulae. In *17th European Conference on Artificial Intelligence*, 2006.

[27] L.J. Stockmeyer. The polynomial-time hierarchy. *Theoretical Computer Science*, 3 :1–22, 1977.

[28] L. Zhang. Solving QBF with combined conjunctive and disjunctive normal form. In *National Conference on Artificial Intelligence (AAAI'06)*, 2006.

Un calcul de Viterbi pour un Modèle de Markov Caché Contraint

Matthieu Petit * **Henning Christiansen**

Research group PLIS : Programming, Logic and Intelligent Systems
Department of Communication, Business and Information Technologies
Roskilde University, P.O Box 260, DK-4000 Roskilde, Denmark
{petit,henning}@ruc.dk

Résumé

Un modèle de Markov caché (MMC) est un modèle statistique permettant de représenter un processus de Markov dont l'état est non observable. Ce modèle a été/est largement utilisé pour adresser des problèmes liés à la reconnaissance vocale ou l'analyse de séquences en bio-informatique. L'algorithme de Viterbi permet de calculer la valeur la plus probable des états cachés du processus étant donné des données observables. Un MMC contraint étend ce cadre de travail en contraignant l'exécution d'un processus d'un MMC. Ce paradigme permet, de manière déclarative, de spécifier le comportement attendu d'un MMC.

Dans cet article, nous définissons un MMC contraint dans le cadre de la Programmation par Contraintes. Nous proposons une nouvelle version de l'algorithme de Viterbi pour ce cadre de travail. Plusieurs techniques issues de la Programmations par Contraintes sont utilisées pour accélérer la recherche de la valeur la plus probable des états cachés du processus. Une implémentation utilisant PRISM, un langage de programmation logique pour des modèles statistiques, est présentée.

Abstract

A hidden Markov model (HMM) is a statistical model in which the system being modeled is assumed to be a Markov process with hidden states. This model has been widely used in speech recognition and biological sequence analysis. Viterbi algorithm has been proposed to compute the most probable value of these hidden states in regards to an observed data sequence. Constrained HMM extends this framework by adding some constraints on a HMM process run.

In this paper, we propose to introduce constrained HMMs into Constraint Programming. We propose new version of the Viterbi algorithm for this new framework. Several constraint techniques are used to reduce the search of the most probable value of hidden states of a constrained HMM. An implementation based on PRISM, a logic programming language for statistical modeling, is presented.

*Ce travail est supporté par le projet "Logic-statistic modeling and analysis of biological sequence data" financé par le programme NABIIT sountenu par "the Danish Strategic Research Council".

1 Introduction

Un modèle de Markov caché (MMC) est un modèle statistique permettant de représenter un processus de Markov pour lequel certains états sont non observables. Cependant, de l'information sur ces états cachés peut être déduite de données observées. En effet, la fréquence d'observation de ces données dépend de l'état du processus dans lequel celles-ci sont émises. Ce modèle a prouvé son utilité pour adresser de nombreux problèmes en reconnaissance vocal [9] et en analyse de séquences biologiques [3]. Cette large utilisation de ce modèle est en partie due aux nombreux algorithmes efficaces permettant :

- le calcul de la séquence la plus probable d'états cachés étant donné des données observées, appelé *chemin de Viterbi* ;
- de faire évoluer le MMC afin de refléter un ensemble de données par apprentissage.

Dans [10], une extension des MMC, appelée modèle de Markov caché contraint, a été proposée par Sato et al.. Ce modèle a pour objectif d'ajouter des contraintes sur les états cachés pouvant être visités et les données émises lors de l'exécution d'un processus de MMC. Cette approche étend l'expressivité de ce modèle en exprimant par des contraintes un ensemble

d'exécutions valides pour un MMC. Par exemple considérons un MMC composé de 100 états cachés. Un MMC contraint permet de spécifier que seulement un dixième de ces états doivent être utilisés lors de l'exécution d'un processus. Dans ce cadre de travail, un algorithme basé sur le calcul de vraisemblance maximale ajusté aux échecs a été proposé dans [10]. Cet algorithme permet par apprentissage de faire évoluer les différents paramètres d'un MMC contraint afin de refléter un ensemble de données observées.

Un certain nombre d'extensions probabilistes de la Programmation par Contraintes ont déjà été proposées. Ces extensions ont pour principal but la modélisation de paramètres incertains dans ce paradigme de programmation [13]. Le caractère aléatoire de ces évènements est modélisé par un choix probabiliste. Dans [4], une probabilité est associée à chaque contrainte d'un problème de satisfaction de contraintes. Cette probabilité caractérise la fréquence avec laquelle la contrainte va être ajouter au problème. La programmation concurrente par contraintes est étendu dans [2, 5] par des opérateurs de choix probabilistes. Ces opérateurs sont utilisés pour modéliser l'aléa dans l'exécution de ce type de processus. Dans [15, 11], Walsh et al. définissent un nouveau cadre de travail, appelé la programmation stochastique à contraintes, permettant de représenter comme un problème à contraintes des problèmes issus de la programmation stochastique. Dans [1], un modèle graphique est proposé par Dechter et al. pour modéliser la recherche de solutions de problèmes à contraintes comportant entre autre des paramètres probabilistes. Récemment dans [8, 7], nous avons défini des contraintes dites de choix probabiliste permettant de raisonner avec des choix probabilistes partiellement définis.

Dans cet article, nous proposons d'introduire les MMC contraints dans le cadre de la Programmation par Contraintes. Cela nous permet de formuler des problèmes classiques associés aux MMC comme des problèmes à contraintes. Plus précisément, nous proposons de modéliser le calcul du chemin de Viterbi comme un problème d'optimisation. Notre objectif est de bénéficier de différentes techniques issus de la Programmation par Contraintes pour effectuer efficacement ce calcul.

Plan de l'article. L'article suit l'organisation suivante : section 2 introduit la notion de MMC contraint à l'aide d'un exemple. La section 3 décrit les pré-requis nécessaires concernant les MCC pour une bonne compréhension de l'article. Dans la section 4, les MCC contraints sont formellement décrits dans le cadre de la Programmation par Contraintes et un calcul de Viterbi est présenté dans ce cadre de travail section 5. Une première implémentation de ce modèle probabiliste, basé sur PRISM, est décrite section 6. Finalement, la section 7 en présentant nos travaux futures à ce sujet conclut l'article.

2 Exemple illustratif

Dans cette section, le cadre de travail des MMC contraints est illustré sur un MMC abstrait. Ce MMC est présenté par FIGURE 1. Le modèle de Markov est composé de quatre états cachés a, b, c et d. Chacun de ces états peut émettre le chiffre 1 ou 2. Les probabilités de transiter d'un état à un autre ou d'émettre 1 ou 2 étant donné un état sont représentées par des arcs étiquetés. Une exécution du processus commence dans l'état *start*, est composée d'une séquence d'états cachés et d'émissions et se termine lorsque l'état *end* est atteint. L'algorithme de Viterbi permet de calculer la séquence la plus probable d'états cachés étant donné une séquence d'émissions connue. Par exemple, le chemin de Viterbi pour la séquence d'émissions 1 1 1 2 est $start\, a\, a\, b\, d\, end$. Un algorithme proposé par Viterbi dans [14] issu de la programmation dynamique permet d'effectuer ce calcul efficacement. Cet algorithme sera détaillé dans la section 3

Des contraintes peuvent être ajoutées sur ce modèle pour spécifier des conditions devant être satisfaites par celui-ci. Par exemple, l'exécution du MMC abstrait peut être contraint à satisfaire : si l'état a a été visité lors de l'exécution, alors l'état d ne peut plus être visité. Afin de représenter cette contrainte, une approche possible consiste à modifier la structure initial du MMC. En effet, en dupliquant les états b et c, il est possible d'interdire pour les chemins ayant transité par a de visiter d. L'approche suivie dans cet article consiste simplement à prendre en compte ces contraintes sur le MMC sans changer la structure initiale du modèle. Le MMC abstrait peut-être étendu par la contrainte que pour toutes exécutions du MMC abstrait $(start\, s_1 \dots s_n\, end\, , e_1 \dots e_n)$

$$s_i = a \implies \forall j > i, s_j \neq d.$$

où $start\, s_1 \dots s_n\, end$ est une séquence d'états cachés et $e_1 \dots e_n$ une séquence d'émissions. Cependant, notons que le calcul Viterbi doit prendre en compte le fait que certaines des exécutions peuvent ne satisfaire pas cette contrainte. Ainsi par exemple, le chemin de Viterbi $start\, a\, a\, b\, d\, end$ calculé pour la séquence d'émissions 1 1 1 2 ne satisfait pas la contrainte énoncée ci-dessus. Pour le MMC abstrait, le chemin de Viterbi pour 1 1 1 2 et satisfaisant la contrainte de ne pas visiter d après avoir visité a est $start\, a\, a\, a\, b\, end$

Les MMC contraints permettent de modéliser différents problèmes tout en conservant la structure initiale du MMC. Il est possible par exemple de contraindre

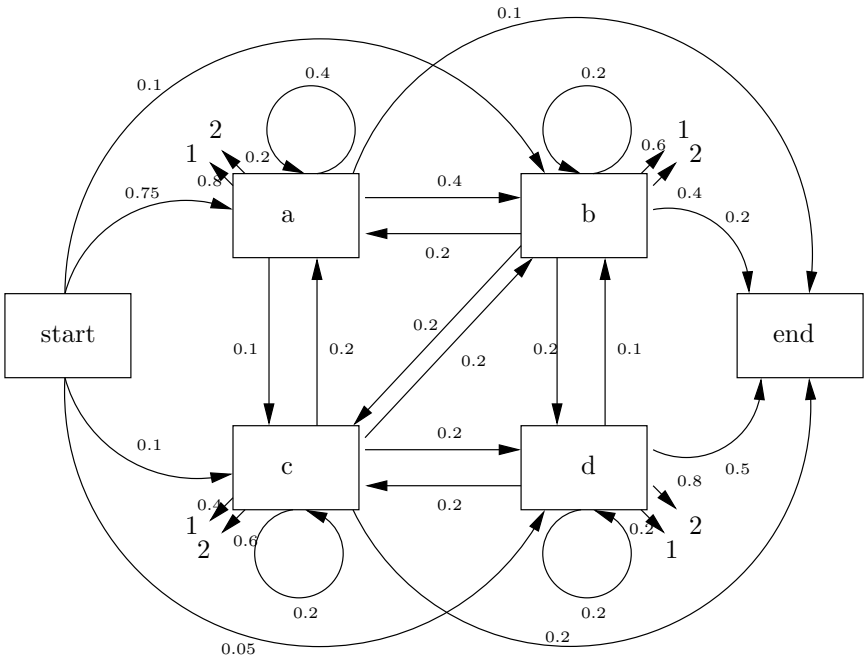


FIGURE 1 – Un MMC abstrait

la somme des émissions à être inférieures à une valeur donnée ou contraindre le nombre de fois qu'un état du modèle est visité à ne pas dépasser une borne maximale. Ces contraintes peuvent exprimer ainsi : pour toutes exécutions $(starts_1 \dots s_n end, e_1 \dots e_n)$

$$\sum_{i=1}^{n} e_i < valeur_1$$

ou

$$cardinality\_atmost(valeur_2, [s_1, \dots, s_n], s).$$

Cependant, ce nouveau cadre de travail nous oblige à adapter les algorithmes classiques associés aux MMC. En effet, ce calcul doit prendre compte le fait que certaines exécutions pouvent ne pas satisfaire le problème à contraintes.

3 Pré-requis sur les Modèles de Markov Cachés

Dans cette section, nous proposons d'introduire les pré-requis nécessaires sur les MMC pour une bonne compréhension du reste de l'article. Une description de l'algorithme de Viterbi est donnée.

3.1 Modèle de Markov Caché

Afin de simplifier la définition formelle de ce modèle, nous nous limitons dans cet article à des MMC n'émettant qu'une seule lettre par transition et n'ayant qu'un unique état initial. La généralisation de ce modèle à des modèles pouvant émettre un nombre arbitraire de lettres (zéro ou plus) et ayant plusieurs états initiaux est simple. L'ordre du processus de Markov est aussi limité au premier ordre et seulement des processus discrets sont considérés.

Définition 1. Un modèle de Markov caché (MMC) est un 4-uplet $\langle S, A, T, E \rangle$ où

- $S = \{s_1, \dots, s_n\}$ est un ensemble d'états parmi lesquels sont distingués : un état initial et des états finaux ;
- A est un ensemble fini de symboles appelés émissions ;
- T est un ensemble de probabilités de transition $\{p(s_i; s_j)\}$ représentant (en excluant l'état initial pour s_j et les états finaux pour s_i) la probabilité de transiter de s_i à s_j. De plus, pour chaque $s_i \in S$, $\sum_{s_j \in S} p(s_i; s_j) = 1$.
- E est un ensemble de probabilités d'émission $\{p(s_i; e_j)\}$ représentant (en excluant l'état initial et les états finaux pour s_i) la probabilité d'émettre e_j de l'état caché s_i. De plus, pour chaque état s_i (en excluant l'étant initial et les états finaux), $\sum_{e_j \in A} p(s_i; e_j) = 1$.

Les données observées lors de l'exécution d'un modèle Markovien caché prennent la forme de séquences d'émissions. Une exécution de ce modèle est composée d'une séquence d'états cachés et d'émissions. Étant donné une séquence d'émissions, une séquence d'états

cachés permettant cette émission est appelée un *chemin*.

Définition 2 (Exécution d'un processus de MMC)**.** Soit $\langle S, A, T, E\rangle$ un modèle de Markov caché, une exécution d'un processus de MMC est composée par une séquence $start\, s_1\, s_2 \ldots s_k\, end$ d'états cachés et une séquence $e_1\, e_2 \ldots e_k$ d'émissions satisfaisant les conditions suivantes :

- $p(start; s_1) \neq 0$ (probabilité de transiter vers s_1 de l'état initial) ;
- $\forall i > 1, p(s_{i-1}; s_i) \neq 0$ (probabilité de transiter entre deux états cachés) ;
- $\forall i, p(s_i; e_i) \neq 0$ (probabilité d'une émission d'un état caché).

Dans le cadre des MMC, plusieurs algorithmes ont été proposés pour donner efficacement une réponse à plusieurs types de questions :

1. Quelle est la probabilité d'observer une séquence d'émissions ?
2. Quel est le chemin le plus probable associé à une séquence d'émissions ?
3. Est-il possible d'adapter les paramètres du MMC afin que ceux-ci reflètent un ensemble de données observées ?

Dans cet article, nous nous focalisons à donner une réponse à la deuxième question dans le cadre des MMC contraints.

3.2 Calcul du chemin de Viterbi pour un MMC

L'algorithme Viterbi [14] est un algorithme issu de la programmation dynamique permettant de déterminer un chemin le plus probable associé à une séquence d'émissions $e_1 \ldots e_k$ pour un MMC. Cet algorithme se base sur un calcul itératif pour chaque état caché s_l du chemin le plus probable atteignant cet état à l'étape i de l'exécution d'un processus Markovien. On note par la suite cette probabilité $p(e_1 \ldots e_i)_{s_l}$. Cette itération est effectuée chaque étape de l'exécution du processus Markovien. L'algorithme est donc de complexité linéaire par rapport la taille de la séquence d'émissions.

Soit $\langle S, A, T, E\rangle$ un modèle de Markov et $e_1 \ldots e_n$ une séquence d'émissions. La probabilité du chemin le plus probable atteignant l'état s_l à l'étape i peut être calculée récursivement selon la formule suivante :

$$p(e_1 \ldots e_i)_{s_l} = \max_{s_k \in S} (p(e_1 \ldots e_{i-1})_{s_k} \cdot p(s_k; s_l) \cdot p(s_l; e_i)) \quad (1)$$

Si l'état caché s_m permet de maximiser $p(e_1 \ldots e_{i-1})_{s_k} \cdot p(s_k; s_l) \cdot p(s_l; e_i)$, le chemin le plus probable atteignant s_l à l'étape i est composé du chemin le plus probable atteignant s_m à l'étape $i-1$ et de la transition entre s_m et s_l.

Basé sur le calcul présenté ci-dessus, l'algorithme de Viterbi est décrit par l'**Algorithme** 1. Il prend comme entrées un MMC $HMM = \langle S, A, T, E\rangle$ et une séquence d'émissions $e_1 \ldots e_n$, et calcule le chemin de Viterbi $Viterbi\_path$ et la probabilité de ce chemin p_{max}.

Algorithme 1 : Algoritme de Viterbi

Entrée : HMM,$e_1 \ldots e_n$
Sortie : $Viterbi\_path$,p_{max}

$(Paths$,$Probas) \leftarrow$ `initialize(`e_1, HMM`)`;
$E \leftarrow \{e_1\, e_2, \ldots, e_1 \ldots e_n\}$;
for $e_1 \ldots e_i \in E$ **do**
  foreach $s_l \in S$ **do**
    $(p(e_i)_{s_l}, s_m) \leftarrow$
    $\max_{s_k \in S}(p(e_1 \ldots e_{i-1})_{s_k} \cdot p(s_k; s_l) \cdot p(s_l; e_i))$
    `update_path(`s_l,s_m,$Paths$,New_Paths`)`;
    `update_proba(`s_l,$Probas$,New_Probas`)`;
  end
  $Paths \leftarrow New\_Paths$;
  $Probas \leftarrow New\_Probas$;
end
$(Viterbi\_path, p_{max}) \leftarrow$
`most_probable(`$Paths$,$Probas$`)`;
return $Viterbi\_path, p_{max}$

L'algorithme Viterbi itère de e_1 à $e_1 \ldots e_n$. Cet algorithme calcule pour chaque état caché du modèle $s_l \in S$ le chemin le plus probable atteignant cet état à l'étape i et la probabilité de ce chemin partiel. Pendant l'itération, les différents chemins ainsi que leurs probabilités respectives sont stockés respectivement dans la structure de données $Paths$ ou $Probas$. Cette structure de données associe à chaque état caché le chemin le plus probable atteignant cet état ou la probabilité de celui-ci. L'équation 1 permet de déduire la valeur de ce chemin ainsi que sa probabilité au regard des valeurs calculées à l'étape précédente de l'itération. La mise à jour des deux structures données est effectuée respectivement par update_path$(s_l, s_m, Paths, New_Paths)$ et update_proba$(s_l, Probas, New_Probas)$. À la fin de l'itération, l'appel à most_probable$(Paths, Probas)$ permet le calcul de p_{max} la probabilité maximale de $Proba$. De l'état caché associé à cette probabilité, le chemin Viterbi $Viterbi\_path$ peut-être déduit.

4 Un modèle de Markov caché contraint

Un MMC contraint a pour objectif de restreindre l'ensemble des exécutions possibles pour un MMC par des contraintes posées sur la séquence d'états cachés

et/ou d'émissions associée à une exécution d'un processus.

Définition 3 (Modèle de Markov Caché Contraint)**.** Soit $\langle S, A, T, E\rangle$ un modèle de Markov caché, un MMC contraint est défini par un 5-tuplet $\langle S, A, T, E, C\rangle$ où C est un ensemble de contraintes associant une exécution du MMC à $\{vrai, faux\}$. Une exécution de MMC contraint R est *valide* si la valeur de $C(R)$ est $vrai$. Sinon, cette exécution est dite *non-valide*.

La définition d'un MMC contraint est volontairement abstraite. Cela nous permet de ne pas avoir à nous placer dans un langage à contraintes particulier. Toutefois, les processus de Markov considérés dans cet article sont discrets. Différentes contraintes issus d'un langage à contraintes sur les domaines finis [12] peuvent par exemple être posées sur l'exécution d'un processus MMC contraint. Notons cependant que l'exécution d'un MMC ne correspond pas à la résolution d'un problème à contraintes classique. En effet, l'ensemble des contraintes posées lors une exécution du modèle dépend des transitions et des émissions composant cette exécution. Par exemple, si on considère la contrainte que l'état d ne peut-être visité si l'état a déjà été visité. La contrainte que d ne peut plus être visité va être posé au cours de l'exécution si a est visité.

Introduire les MMC contraints dans le cadre de la Programmation par Contraintes autorise à ne pas devoir décrire explicitement l'ensemble des exécutions valides et non-valides du modèle. Une exécution valide ou non-valide est détectée par la résolution du problème à contraintes associé à celle-ci. En contrepartie, ce cadre de travail introduit une nouvelle complexité dans le calcul de Viterbi certains chemins pouvant ne pas satisfaire le problème à contraintes. Afin de modéliser ce calcul comme un problème classique de recherche de solutions, nous proposons de modéliser cette recherche comme une recherche dans un arbre, appelé arbre probabiliste.

Définition 4 (Arbre probabiliste)**.** Soit $\langle S, A, T, E, C\rangle$ un MMC contraint. Un arbre probabiliste est 4-uplet $\langle Noeuds, Arcs, CP, Probabilitées\rangle$. Un élément de $Noeuds$ représente un couple état caché émission (en excluant le noeud initial et les noeuds finaux). Un élément de $Arcs$ représente une transition vers un état caché ainsi que l'émission produite de cet état. Chaque noeud N de $Noeuds$ est étiqueté par un problème de satisfaction de contraintes (CSP) composant CP. Chaque variable de ce CSP représente les possibles exécutions du modèle Markovien passant par ce noeud. Chaque arc N_iN_{i+1} de $Arcs$ est étiqueté par la probabilité de transiter de N_i à N_{i+1}.

Un chemin de l'arbre allant de la racine à une feuille correspond à l'exécution d'un MMC. La probabilité de cette exécution est calculée en multipliant les différentes probabilités associées aux arcs composant le chemin. L'exécution du MMC contraint est valide si le CSP associé à la feuille atteinte par cette exécution est satisfaisable.

Dans la suite, seule une version de l'arbre dédiée au calcul de Viterbi est considérée. En effet, les données émises sont connues pour ce calcul. Donc, étant donné une séquence d'émissions, seule une restriction de l'arbre probabiliste pour cette séquence sera considérée. De plus, comme la séquence d'émissions est de taille finie, cela nous permet de raisonner sur un arbre de recherche de taille finie. Cette restriction d'un arbre probabiliste PT à une séquence d'émissions $e_1 \dots e_n$ est notée $PT_{|e_1 \dots e_n}$.

Comme exemple, la FIGURE 2 représente les deux premières étapes de l'arbre probabiliste du MMC abstrait contraint à satisfaire : pour toutes exécutions du MCC $(start\, s_1 \dots\ s_n end, e_1 \dots e_n)$

$$s_i = a \implies \forall j > i, s_j \neq d.$$

étant la séquence d'émissions 1 1 1 2. Dans cet arbre, les noeuds sont étiquetés par des contraintes posées sur les différents états cachés représentant les exécutions possibles du processus. Des variables S_1, S_2, S_3 et S_4 sont utilisées pour représenter l'état caché choisi à l'étape 1, 2, 3 et 4. Dans ce contexte, une affectation de ces variables représente le choix d'un chemin. Le calcul du chemin de Viterbi pour les MMC contraints doit chercher une affectation de ces variables la plus probable satisfaisant le problème à contraintes associées à cette affectation.

5 Calcul de Viterbi pour les MMC contraints

L'algorithme présenté dans la sous-section 3.2 permet le calcul du chemin de Viterbi pour un MMC classique. Ce calcul est effectué linéairement par rapport à la taille de la séquence d'émissions. Basé sur l'équation 1, le calcul du chemin de Viterbi ne nécessite que de se remémorer des différents chemins les plus probables atteignant chaque état caché pour chaque étape du processus. Cependant, le cadre des MMC contraints introduit une nouvelle complexité dans ce calcul. En effet, il est nécessaire de prendre compte la validité du chemin suivi, i.e. prendre compte la satisfiabilité des contraintes associées à ce chemin. Dans le pire des cas, le calcul de Viterbi pour une séquence d'émissions $e_1 \dots e_n$ dans le contexte des MMC contraints nécessitent de tester la satisfiabilité des CSP associés à chacun des chemins $PT_{|e_1 \dots e_n}$ et de choisir le

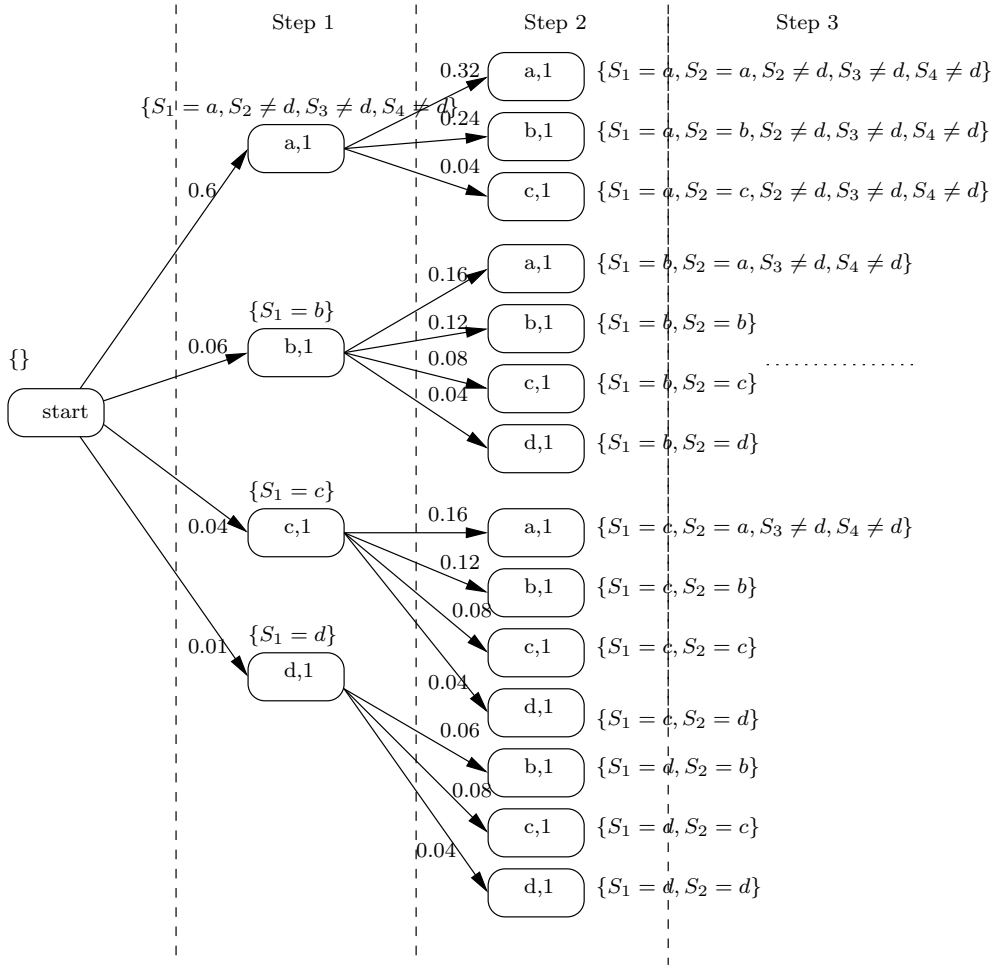


FIGURE 2 – Arbre probabiliste associé à l'observation 1 1 1 2 pour le MMC abstrait

plus probable parmi ceux-ci. Toutefois, l'introduction de ce modèle dans le cadre de la Programmation par Contraintes nous permet de bénéficier de différentes techniques existantes permettant de réduire l'espace de recherche associé au calcul du chemin de Viterbi pour les MMC contraints. Dans cette section, nous proposons donc d'exprimer le calcul de Viterbi comme un problème d'optimisation et utilisons certaines techniques permettant de réduire quand cela est possible la complexité de ce calcul.

5.1 Énoncé du problème

Considérons un MMC contraint $\langle S, A, T, E, C\rangle$, et une séquence d'émissions $e_1 \dots e_n$. Le calcul du chemin de Viterbi a pour objectif de déterminer la séquence d'états cachés $start\, s_1 \dots s_n end$ pour laquelle l'exécution $(start\, s_1 \dots s_n\, end, e_1 \dots e_n)$ est la plus probable et tel que le CSP associé à cette exécution soit satisfaisable.

Dans les termes de la Programmation par Contraintes, le calcul du chemin de Viterbi consiste à chercher une affectation $(s_1, \dots, s_n)$ des variables $S_1, \dots, S_n$ à valeurs dans $S \times \dots \times S$ tel que

$$\begin{array}{rccl} f: & S \times \dots \times S & \rightarrow & [0;1[\\ & (s_1, \dots, s_n) & \mapsto & p(start; s_1) \cdot p(s_1; e_1) \dots \\ & & & p(s_n; e_n) \cdot p(s_n; end) \end{array}$$

soit maximisée et le problème à contraintes associé à ce chemin soit satisfaisable.

5.2 Algorithme

L'algorithme de recherche du chemin de Viterbi pour un MMC contraint est donné par l'**Algorithme**2. Il a pour entrée un modèle de Markov caché contraint $Ctr\_HMM = \langle S, A, T, E, C\rangle$ et une séquence d'émissions $e_1 \dots e_n$, et calcule le chemin de Viterbi $Viterbi\_path$ et la probabilité de ce chemin p_{max}.

Algorithme 2 : Algorithme de recherche du chemin de Viterbi

Entrée : Ctr_HMM,$e_1 \dots e_n$
Sortie : $Viterbi\_path$,p_{max}

$p_{max} = 0$;
$hs \leftarrow [S_1, S_2, \dots, S_n]$;
`search_Viterbi`(hs,pr,$partial\_path$) {
if $hs = \emptyset$ **then**
  $p_{max} \leftarrow pr$;
  $Viterbi\_path \leftarrow partial\_path$;
end
$S_i \leftarrow$`first_element`(hs);
foreach $s \in \mathrm{dom}(S_i)$ **do**
  $s_p \leftarrow$ `previous_state`($partial\_path$);
  $pr \leftarrow pr * p(s_p; s) * p(s; e_i)$;
  if $pr > p_{max}$ **then**
    $partial\_path \leftarrow$`add`($partial\_path, s$);
    if `check_sat`($C(partial\_path, e_1 \dots e_i)$) **then**
      $hs \leftarrow$ `remove`(S_i, hs);
      `search_Viterbi`(hs,pr,$partial\_path$)
    end
  end
end
}

Le calcul est basé sur un appel itératif de $search\_Viterbi$. Cette itération permet le parcours en profondeur d'abord de l'arbre probabiliste associé à Ctr_HMM étant donné la séquence d'émissions $e_1 \dots e_n$. Ce parcourt prend la forme de l'affectation successive des variables S_i représentant le choix d'un état caché à l'étape i. L'affection partielle de ces variables est stockée dans $partial\_path$. La recherche est effectuée pour chaque valeur possible de la variable S_i. La variable pr est utilisée pour stocker la probabilité d'atteindre s à l'étape i. Le parcours de l'arbre peut-être interrompu dans deux situations :

- quand la probabilité de sélectionner ce chemin partiel est inférieure à la courante meilleure solution ;
- quand le problème à contraintes associé au noeud atteint est détecter comme non-satisfaisable.

La variable p_{max} est utilisée pour se remémorer de la courant meilleur solution. Le test de la satisfiabilité du CSP associé au noeud atteint est effectué par une procédure de décision représentée par l'appel de check_sat. Lorsque l'ensemble des variables hs réduit à $\emptyset$, alors une feuille de l'arbre est atteinte. Dans ce cas, une nouvelle solution optimale vient d'être trouvée et les variables $Viterbi\_path$ et p_{max} sont donc mises à jour.

Terminaison et complexité.

Comme l'ensemble des variables est fini, la recherche effectuée dans l'arbre probabiliste termine comme l'arbre est d'hauteur et de largeur finie. Cependant dans le pire des cas, l'algorithme doit tester la satisfiabilité de chacun des CSP associé aux feuilles de l'arbre. Or le nombre de ces feuilles croit exponentiellement par rapport à la longueur de la séquence d'émissions. Cependant, les techniques de cohérences partielles permettent de détecter dès que possible que le chemin partiel ne menant à aucune exécution du modèle valide.

5.3 Réduction de l'arbre par partage de sous-arbre

Par définition, un processus de Markov n'a pas besoin d'avoir une information complète du passé pour effectué une transition. Dans le cadre dans lequel nous nous sommes placés, celui des MMC du premier ordre, la probabilité de transition dépend uniquement de l'état caché atteint à l'étape précédente. En regardant l'arbre probabiliste associé à un MMC contraint, cela peut se traduire par des sous-arbres similaires issus d'un même état caché pour une étape donnée du processus. Considérons de nouveau l'arbre probabiliste $PT_{|1\,1\,1\,2}$, on peut remarquer que les CSP associés respectivement au chemin partiel $start\,b\,a$ et $start\,c\,a$ sont identiques. Cette affirmation nous permet donc de déduire que les sous-arbres associés à ces deux noeuds seront les mêmes. Or la probabilité de sélectionner le chemin partiel $start\,b\,a$ est de $0,0096$ alors que la la probabilité de sélectionner le chemin partiel $start\,c\,a$ est de $0,0064$. Comme le calcul de Viterbi recherche le chemin le plus probable, cela veut dire que le sous-arbre issu du chemin partiel ayant la plus faible probabilité ($startca$) ne va pas mener à une solution optimale.

Plus formellement, considérons l'arbre probabiliste $PT_{|e_1 \dots e_n}$, un chemin partiel de cet arbre $start\,s_1\,s_2 \dots s_j$ (resp. $start\,s'_1\,s'_2 \dots s_j$) atteignant le noeud N_i (resp. N'_i), C_i (resp. C'_i) le problème à contraintes associé à ce noeud et p_i (resp. p'_i) la probabilité d'atteindre ce noeud. Notons par P et P' les CSPs $\langle \{s_1, \dots, s_j, S_{j+1}, \dots, S_n\}, S \times \dots \times S, C_i \rangle$ and $\langle \{s'_1, \dots, s'_j, s_j, S_{j+1}, \dots, S_n\}, S \times \dots \times S, C'_i \rangle$. Alors, si

$$\mathrm{sol}(P') \subseteq \mathrm{sol}(P) \text{ et } p_i \geq p'_i$$

où sol est une fonction associant à un CSP l'ensemble de ces solutions, alors le sous-arbre associé au noeud N'_i peut-être retiré de l'arbre de recherche. Notons que la recherche d'inclusion de solutions de deux sous-CSPs est un problème difficile et coûteux à résoudre dans le cas général. Cependant pour certains sous-ensemble de problèmes à contraintes, il est possible de mettre en place des tests d'inclusion efficaces adaptés au MMC contraint considéré.

6 Implémentation

Dans cette section, nous montrons que le langage de programmation PRISM permet d'implémenter le cadre des MMC contraints. En effet, l'implémentation de PRSIM est basée sur B-Prolog. Il nous permet de combiner les différents résolveurs de contraintes de B-Prolog avec PRISM. Nous illustrons cette implémentation sur l'exemple du MMC abstrait donné FIGURE 1.

6.1 Une brève introduction à PRISM

PRISM [10] est un système développé par Sato et al. permettant dans un cadre déclaratif la définition de différents modèles probabiliste-logiques. Ce système étend Prolog par l'ajout de variables aléatoires discrètes. La déclaration

```
values(lettre, [a,c,g,t])
```

permet la création de la variable aléatoire `lettre` à valeurs dans `[a,c,g,t]`. Par défaut, cette variable aléatoire a une distribution uniforme. Un appel `msw(lettre,X)` pendant l'exécution d'un programme correspond au choix aléatoire d'une valeur pour `X` selon la définition de `lettre`.

La déclaration des variables aléatoires peut être paramétrée, comme

```
values(lettre(_), [a,c,g,t,*])
```

qui est un raccourci pour définir un nombre infini de variables aléatoires dont le nom peut s'unifier au terme `letter(_)`. Cet ensemble de déclarations peut-être utilisé pour définir une transition d'un état à un autre pour un modèle de Markov simple.

```
s(S):- s(-,S).
s(*,[]).
s(X,[L|Ls]):-
      X\= *,
      msw(letter(X),L),
      s(L,Ls).
```

Dans ce modèle, '-' est utilisé pour représenter le début de l'exécution du processus et '*' la fin.

Un programme PIRSM peut-être utilisé de différentes manières. Le système comprend entre autre des prédicats permettant la génération d'échantillon d'exécutions du modèle probabiliste. Ce langage inclut aussi des algorithmes d'apprentissage permettant de faire évoluer les paramètres d'un modèle afin de représenter le plus vraisemblablement un ensemble de données. Pour finir, une version généralisée de l'algorithme de Viterbi au modèle probabiliste-logique est disponible. Cela permet entre autre d'effectuer ce calcul pour différents modèles probabilistes comme les modèles de Markov cachés, les réseaux bayésien discret ...

6.2 Une exemple de MMC contraint avec PRISM

Cette sous-section présente l'implémentation du MMC abstrait donné par la FIGURE 1. Ce modèle est contraint à safistaire la condition suivante : pour chaque exécution du MMC $(start\, s_1 \dots s_n\, end, e_1 \dots e_n)$

$$s_i = a \implies \forall j > i, s_j \neq d.$$

Le code suivant représente le programme PRISM définissant ce MMC contraint.

```
% Transition état-état
values(start,[a,b,c,d]).
values(trans(_),[a,b,c,d,end]).

% Transition état-émission
values(emit(_),[1,2]).

% Définition des distributions de probabilités
set_params:-
   set_sw(begin,[0.75,0.1,0.1,0.05]),
   set_sw(trans(a),[0.4,0.4,0.1,0,0.1]),
   set_sw(trans(d),[0,0.1,0.2,0.2,0.5]),
   set_sw(emit(a),[0.8,0.2]),
   set_sw(emit(b),[0.6,0.4]),
   set_sw(emit(c),[0.4,0.6]),
   set_sw(emit(d),[0.2,0.8]).

% Initialisation du Processus
abstract_ctr(Emissions) :-
   set_params,
   msw(begin,State), % Choix du 1er état
   SeenA in 0..1,
   abstract_ctr(State,SeenA,Emissions).

% Fin de l'exécution
abstract_ctr(end,_SeenA,[]).

% Etat a non visité
abstract_ctr(State,SeenA,[Letter|Rest]) :-
   State \== end,
   var(SeenA), % a non visité
   msw(emit(State),Letter), % Emission
   msw(trans(State),New_State),% Transition
   New_State #= a # => SeenA#=1
   abstract_ctr(State_New,SeenA,Rest).

% Etat a visité
abstract_ctr(State,SeenA,[Letter|Rest]) :-
   State \== end,
   msw(emit(State),Letter) % Emission
   msw(trans(State),New_State),% Transition
   SeenA #= 1 #=> New_State #\=d),
   abstract_ctr(State_New,SeenA,Rest).
```

Une exécution du MMC contraint a lieu par l'appel au prédicat `abstract_ctr/1`.

```
?- abstract_ctr(S).
```

Une exécution valide a été suivie lorsque la requête réussit

```
?- abstract_ctr(S).
S = [1,1,1,2]
yes
```

Cette requête échoue lorsque le chemin choisi ne satisfait pas la contrainte :

```
?- abstract(S).
false
```

Les transitions entre états cachés du modèle sont modélisées par l'appel à

```
msw(trans(State),New_State)
```

La génération d'une émission étant donné un état caché est effectuée à l'aide de

```
msw(emit(State),Letter)
```

La variable `SeenA` est utilisée pour savoir si l'état caché `a` a déjà été visité. Le système PRISM permet le calcul du chemin de Viterbi pour ce type modèle par l'appel de

```
?- viterbi(abstract_ctr([1,1,1,2]),P,Path).
P = 0.00098304
Path = [start,a,a,a,b,end]
yes
```

Dans cette implémentation, nous utilisons l'algorithme de Viterbi générique de PRISM pour différents modèles probabilistes. Des techniques de cohérences d'intervalles utilisés par le résolveur de contraintes sur les domaines finis de B-Prolog permettent réduire dès que possible l'arbre de recherche. Toutefois, cette implémentation n'intègre pas la technique le partage de sous-arbres proposée sous-section 5.3.

7 Travaux futurs

Dans cet article, nous avons proposé les MMC contraints dans le cadre de la Programmation Contraintes. Cette approche nous permet de modéliser le calcul du chemin de Viterbi comme un problème d'optimisation. L'introduction de ce modèle s'est voulue générique, au sens où ce modèle n'est pas restreint à un type particulier de résolveur de contraintes. Une première implémentation basée sur le système PRISM et utilisant le résolveur de contraintes sur les domaines finis de B-Prolog est proposée. Cette implémentation offre un cadre déclaratif pour la définition de MMC contraints. Toutefois, de nombreuses améliorations peuvent être apportées à l'algorithme proposé. En restant dans le cadre de la programmation à contraintes sur les domaines finis, ce calcul de Viterbi pour les MMC contraints peut bénéficier de nombreux résultats obtenus dans le cadre de la recherche And/Or pour des modèles graphiques [1] et utiliser les différents algorithmes de cohérence locale pour les CSP pondérés [6]. La réduction de l'arbre de recherche par partage de sous-arbre est un problème intéressant que nous désirons développer.

Références

[1] R. Dechter and R. Mateescu. And/or search spaces for graphical models. *Artificial Intelligence*, 171(2-3) :73–106, February 2007.

[2] A. Di Pierro and H. Wiklicky. On probabilistic CCP. In *Proceedings of the Joint Conference on Declarative Programming*, pages 225–234, Grado, Italy, 1997.

[3] R. Durbin, S.R. Eddy, A. Krogh, and G. Mitchison. *Biological Sequence Analysis : Probabilistic Models of Proteins and Nucleic Acids*. Cambridge University Press, 1998.

[4] H. Fargier and J. Lang. Uncertainty in constraint satisfaction problems : a probabilistic approach. In *Proceedings of the European Conference on Symbolic and Quantitative Approaches to Reasoning and Uncertainty*, LNCS, pages 97–104, Granada, Spain, November 1993.

[5] V. Gupta, R. Jagadeesan, and V.A. Saraswat. Probabilistic concurrent constraint programming. In *Proceedings of the International Conference Conference on Concurrency Theory*, LNCS, pages 243–257, Warsaw, Poland, 1997. Springer.

[6] J. Larrosa and T. Schiex. Solving weighted CSP by maintaining arc consistency. *Artificial Intelligence*, 159(1–2) :1–26, 2004.

[7] M. Petit. *Test statistique structurel par résolution de contraintes de choix probabiliste (in french)*. Phd thesis, Université de Rennes 1, France, Juillet 2008.

[8] M. Petit and A. Gotlieb. Boosting probabilistic choice operators. In *Proceedings of the International Conference on Principles and Pratice of Constraint Programming*, LNCS, pages 559–573, Providence, USA, September 2007.

[9] L.R. Rabiner. A tutorial on hidden markov models and selected applications in speech recognition. *Proceedings of IEEE*, 77(2) :257–286, 1989.

[10] T. Sato and Y. Kameya. New advances in logic-based probabilistic modeling by prism. In *Probabilistic Inductive Logic Programming*, pages 118–155, 2008.

[11] S. A. Tarim, S. Manandhar, and T. Walsh. Stochastic constraint programming : a scenario-based approach. *Constraints*, 11(1) :53–80, 2006.

[12] P. Van Hentenryck, V.A. Saraswat, and Y. Deville. Design, implementation, and evaluation of the constraint language cc(fd). *Constraint Programming*, 910 :293–316, May 1995.

[13] G. Verfaillie and N. Jussien. Constraint solving in uncertain and dynamic environments : A survey. *Constraints*, 10(3) :253–281, July 2005.

[14] A.J. Viterbi. Error bound for convutional codes and an asymptotically optimum algorithm. *Transactions on Information Theory*, 13(2) :260–269, 1967.

[15] T. Walsh. Stochastic constraint programming. In *Proceedings of the European Conference on Artificial Intelligence*, pages 111–115, Lyon, France, July 2002. IOS Press.

Réordonnancement dynamique basé sur l'apprentissage

Youssef Hamadi[1] **Said Jabbour**[2] **Lakhdar Sais**[2]

[1] Microsoft Research
7 J J Thomson Avenue
Cambridge, United Kingdom
youssefh@microsoft.com

[2] CRIL-CNRS, Université d'Artois
rue Jean Souvraz SP18
F-62307 Lens Cedex France
{jabbour,sais}@cril.fr

Abstract

Cet article propose un nouveau schéma d'apprentissage. La particularité de cette approche réside dans sa capacité à opérer sans se référer aux conflits. Ceci contraste avec les approches d'apprentissage traditionnelles qui sont généralement basées sur l'analyse de conflits. Pour permettre un tel apprentissage, des clauses pertinentes issues de la partie satisfaites de la formule sont conjointement combinées au graphe d'implications classique afin de générer une meilleure raison d'implication pour un littéral donné. A partir de cette extension, un premier schéma d'apprentissage appelé réordonnancecement dynamique basé sur l'apprentissage (RDBA) est proposé. Il exploite les nouvelles raisons générées pour réordonner dynamiquement les affectations partielles. Des résultats expérimentaux montrent que l'intégration de RDBA sur des solveurs issus de l'état de l'art de SAT permet d'obtenir d'intéressantes améliorations, notamment sur les instances satisfiables.

1 Introduction

Le problème SAT, qui consiste à vérifier si un ensemble de clauses est satisfiable, est central dans plusieurs domaines de l'informatique. Il est important en intelligence artificielle, en satisfaction de contraintes (CSP), planification, raisonnement non monotone, vérification de circuits, etc. SAT est devenu un domaine attractif. Ceci s'explique par l'apparition d'une nouvelle génération de solveurs appelés solveurs SAT modernes [9, 4] particuliérement efficaces. Ces solveurs sont basés sur la propagation unitaire [2] combinée efficacement à des structures de données comportant : (i) des stratégies de redémarrage [6, 7], (ii) des heuristiques de choix de variables (comme VSIDS) [9], et (iii) l'apprentissage de clauses [8, 9]. Les solveurs SAT modernes sont spécialement efficaces sur les instances issues d'applications industrielles. Sur ces problèmes, Selman et al. [5] ont identifiés le phénomène de longue traîne (heavy tailed), i.e., diffèrents ordres de variables mènent souvent à des temps de résolution différents. Ceci explique entre autre l'introdution des stratégies de redémarrage dans les solveurs SAT modernes, dans le but de découvrir un bon ordre des variables. Par ailleurs, VSIDS et d'autres variantes d'heuristiques de choix de variables ont été introduites pour éviter le trashing et focaliser la recherche : lorsqu'il s'agit d'instances de grande taille ces heuristiques tentent de diriger la recherche vers la partie la plus contrainte de la formule. Les redémarrages et VSIDIS jouent un rôle complémentaire dès lors qu'ils implémentent respectivement les deux principes de diversification et d'intensification. L'apprentissage dirigé par les conflits (Conflict Driven Clause Learning (CDCL)) est le troisième composant, conduisant au retour-arrière non-chronologique. Dans CDCL une structure de données centrale est le *graph d'implications*, qui mémorise l'interprétation partielle en cours de construction ainsi que les implications faites. À chaque fois qu'un échec est rencontré, une clause conflit ou nogood est apprise grâce au parcours par le bas du graphe d'implications. Ce parcours est aussi utilisé pour mettre à jour les activités des variables concernées, permettant à l'heuristique VSIDS de sélectionner en priorité la variable la plus active lors de la prochaine décision. Dans ce papier nous traitons de l'apprentissage dans les solveurs SAT modernes. Notre but est de montrer que de façon similaire à la vie réelle, on peut apprendre non seulement à partir des échecs (ou conflits) mais aussi à partir des décisions menant à des succès. Comme l'apprentissage renvoi classiquement à l'analyse de conflits, le cadre que nous proposons contraste avec les approches classiques d'apprentissage proposées dans le cadre SAT.

Dans l'apprentissage classique, le but principale est de calculer la raison du conflit courant et de permettre d'effectuer un retour-arrière non chronologique. À l'opposé, notre approche a pour but principal de déduire de meilleures raisons pour l'implication d'un littéral donné permettant d'exploiter ces raisons pour réarranger (ou corriger) l'affectation partielle courante. Usuellement, à un nœud donné de l'arbre de recherche, lorsqu'un littéral de décision est affecté et après application de la propagation unitaire, la raison enregistrée pour un littéral impliqué x contient au moins un littéral du niveau courant i en plus du littéral x. Cette raison est codée dans le graphe d'implications. L'approche que nous proposons tend à découvrir de nouvelles raisons pour x, incluant seulement des littéraux de niveaux ($j < i$). Autrement dit, notre approche est capable de découvrir qu'un littéral propagé au niveau i aurait dû être propagé plus tôt, au niveau j du processus de recherche. En pratique, ces nouvelles raisons lorsqu'elles sont codées dans le graphe d'implication pourraient améliorer l'analyse de conflits elle-même. e.g., en améliorant les niveaux des retours-arrière. Contrairement à l'analyse de conflits classique où le littéral assertif est affecté au niveau du retour-arrière à sa valeur de vérité opposée (littéral refuté), dans l'approche que nous proposons, les nouvelles raisons générées sont utilisées pour réarranger l'interprétation courante en affectant quelques littéraux aux niveaux précédents en gardant les mêmes valeurs de vérité. En considérant le niveau d'implication de quelques littéraux, notre approche tend à repousser les mauvaises décisions en bas de l'arbre de recherche. Ce processus améliore la qualité de l'affectation partielle courante et favorise l'obtention des solutions. Le papier est organisé comme suit. Après quelques notations et définitions préliminaires, nous présentons le graphe d'implications et le schéma d'apprentissage classiques section 2.1 ensuite nous décrivons notre approche dans la section 3. Dans la section 4, l'intégration de RDBA au sein des solveurs SAT modernes est présentée. Finalement, avant la conclusion, des résultats expérimentaux quantifiant les performances de notre approche sont présentées.

2 Definitions

2.1 Definitions préliminaires et notations

Une *formule CNF* $\mathcal{F}$ est un ensemble (interprété comme une conjonction) de *clauses*, où chaque clause est un ensemble (interprété comme une disjonction) de *littéraux*. Un littéral est soit positif (x) ou négatif ($\neg x$). Les deux littéraux x et $\neg x$ sont appelés *complémentaires*. On note également $\neg l$ (ou $\bar{l}$) le littéral complémentaire de l. Pour un ensemble de littéraux L, $\bar{L}$ désigne l'ensemble $\{\bar{l} \mid l \in L\}$. Une clause est dite *unitaire* si elle contient exactement un seul littéral (appelé *littéral unitaire*), et dans le cas d'une clause contenant seulement deux littéraux celle-ci est dite binaire. La *clause vide*, notée $\bot$, est interprétée comme fausse (insatisfaisable), tandis qu'une *formule CNF vide*, notée $\top$, est interprétée comme vraie (satisfaisable). L'ensemble des littéraux apparaissant dans $\mathcal{F}$ est noté $V_{\mathcal{F}}$. Une *interprétation* ρ d'une formule booléenne $\mathcal{F}$ associe une valeur $\rho(x)$ aux variables $x \in \mathcal{F}$. L'interprétation ρ est dite *complète* si elle associe une valeur à chaque $x \in \mathcal{F}$, sinon elle est dite *partielle*. Une interprétation peut également être représentée par un ensemble de littéraux. Un *modèle* d'une formule $\mathcal{F}$ est une interprétation ρ qui satisfait la formule, ce qui est noté $\rho \vDash \mathcal{F}$. Les notations suivantes seront largement utilisés par la suite :

- $\mathcal{R}[x, c_i, c_j]$ dénote la *résolvante* entre la clause c_i contenant le litéral x et la clause c_j contenant l'opposé de ce même litéral $\neg x$. Autrement dit $\mathcal{R}[x, c_i, c_j] = c_i \cup c_j \backslash \{x, \neg x\}$. Une résolvante est appelé *tautologique* si elle contient à la fois un litéral et son opposé.
- $\mathcal{F}|_x$ dénote la formule $\mathcal{F}$ simplifiée par l'affectation du litéral x à la valeur *vrai*. Formellement $\mathcal{F}|_x = \{c \mid c \in \mathcal{F}, \{x, \neg x\} \cap c = \emptyset\} \cup \{c \backslash \{\neg x\} \mid c \in \mathcal{F}, \neg x \in c\}$ (les clauses contenant x et qui sont donc satisfaites sont supprimées ; et celles contenant $\neg x$ sont simplifiées). Cette notation est étendue aux interprétations : soit $\rho = \{x_1, \ldots, x_n\}$ une interprétation, on définit $\mathcal{F}|_\rho = (\ldots((\mathcal{F}|_{x_1})|_{x_2})\ldots|_{x_n})$.
- $\mathcal{F}^*$ dénote la formule close $\mathcal{F}$ après application de la propagation unitaire, définie récursivement comme suit : (1) $\mathcal{F}^* = \mathcal{F}$ si $\mathcal{F}$ ne contient aucune clause unitaire. (2) $\mathcal{F}^* = \bot$ si $\mathcal{F}$ contient deux clauses unitaires $\{x\}$ et $\{\neg x\}$, (3) autrement, $\mathcal{F}^* = (\mathcal{F}|_x)^*$ tel que x est le litéral qui apparaît comme clause unitaire dans $\mathcal{F}$.
- $\models_*$ dénote la déduction logique par propagation unitaire : $\mathcal{F} \models_* x$ signifie que le litéral x est déduit par propagation à partir de $\mathcal{F}$, i.e. $x \in \mathcal{F}^*$. On écrit $\mathcal{F} \models_* \bot$ si la formule est inconsistance (insatisfaisable) par propagation. En particulier, si on dispose d'une clause c tel que $\mathcal{F} \wedge \overline{c} \models_* \bot$, alors c est une conséquence logique de $\mathcal{F}$. On dit alors que c est déduite *par réfutation*.

2.2 Recherche DPLL

Nous introduisons maintenant des notations et remarques terminologiques associées aux solveurs SAT basés sur la procédure de Davis Logemann Loveland, communément appelé DPLL [2]. DPLL est une procédure de recherche de type "*backtrack*" ; À chaque nœud les littéraux affectés (le litéral de décision et les littéraux propagés) sont étiquetés avec le même *niveau de décision*, initialisé à 1 et incrémenté à chaque nouveau point de décision. Le niveau de décision courant est le niveau le plus élevé dans la pile de propagation. Lors d'un retour-arrière ("*backtrack*"), les variables ayant un niveau supérieur au niveau du backtrack

sont défaites et le niveau de décision courant est décrémenté en consèquence (égal au niveau du backtrack). Au niveau i, l'interprétation partielle courante ρ peut être représentée comme une séquence décision–propagations de la forme $\langle (x_k^i), x_{k_1}^i, x_{k_2}^i, \ldots, x_{k_{n_k}}^i \rangle$ telle que le premier littéral x_k^i correspond au littéral de décision x_k affecté au niveau i et chaque $x_{k_j}^i$ de l'ensemble $1 \leq j \leq n_k$ représente les litéraux unitaires propagés à ce même niveau i.

Soit $x \in \rho$, On note $l(x)$ le niveau d'affectation de x, $d(\rho, i) = x$ si x est le littéral de décision affecté au niveau i. Pour un niveau donné i, ρ^i représente la projection de ρ aux litéraux affectés au niveau $\leq i$.

2.3 Analyse du Conflit et Graphe d'Implication

Les graphes d'implications sont des représentations capturant les variables affectées durant la recherche, que ce soit des variables de décision ou des variables propagées. Cette représentation est utile pour analyser les conflits. À chaque fois qu'un littéral y est propagé, on sauvegarde la clause à l'origine de cette propagation, clause que l'on note $\overrightarrow{imp}(y)$. La clause $\overrightarrow{imp}(y)$ est donc de la forme $(x_1 \vee \cdots \vee x_n \vee y)$ où chaque littéral x_i est faux sous l'interprétation courante ($\forall i \in 1 \ldots n \quad \rho(x_i) = \textit{faux}$) et $\rho(y) = \textit{vrai}$. Quand un littéral y n'est pas obtenu par propagation unitaire mais qu'il correspond à un point de choix, $\overrightarrow{imp}(y)$ est indéfini, que l'on note par convention $\overrightarrow{imp}(y) = \bot$.

Lorsque $\overrightarrow{imp}(y) \neq \bot$, on note par $exp(y)$ l'ensemble $\{\overline{x} \mid x \in \overrightarrow{imp}(y) \setminus \{y\}\}$, appelé l'ensemble des *explications* de y. Autrement dit, si $\overrightarrow{imp}(y) = (x_1 \vee \cdots \vee x_n \vee y)$ alors les explications sont les litéraux $\overline{x_i}$ qui constituent la condition sous laquelle $\overrightarrow{imp}(y)$ est devenue une clause unitaire $\{y\}$. Notons que pour tout i on a $l(\overline{x_i}) \leq l(y)$, i.e., toutes les explications de la déduction ont un niveau inférieur à celui de y.

Quand $\overrightarrow{imp}(y)$ est indéfini, $exp(y)$ est égal à l'ensemble vide. Les explications peuvent, alternativement, être vues comme un graphe d'implications dans lequel l'ensemble des prédécesseurs d'un nœud correspond aux explications du littéral correspondant :

Définition 1 (Graphe d'implication) *Soient $\mathcal{F}$ une formule CNF et ρ une interprétation partielle. Le graphe d'implication associé à $\mathcal{F}$, ρ et exp est $(\mathcal{N}, \mathcal{E})$ tel que :*

- *$\mathcal{N} = \rho$, i.e., there is exactly one node for every literal, decided or implied ;*
- *$\mathcal{E} = \{(x, y) \mid x \in \rho, y \in \rho, x \in exp(y)\}$*

Par souci de simplicité, exp est éliminé dans le reste du papier, et le graphe d'implication est tout simplement noté $\mathcal{G}_{\mathcal{F}}^{\rho}$.

Exemple 1 *$\mathcal{G}_{\mathcal{F}}^{\rho}$, montré dans la Figure 1 est le graphe d'implications de la formule $\mathcal{F}$ et l'affectation partielle ρ donnée ci-dessous : $\mathcal{F} \supseteq \{c_1, \ldots, c_9\}$*

$(c_1)\ x_6 \vee \neg x_{11} \vee \neg x_{12}$ $\quad$ $(c_2)\ \neg x_{11} \vee x_{13} \vee x_{16}$
$(c_3)\ x_{12} \vee \neg x_{16} \vee \neg x_2$ $\quad$ $(c_4)\ \neg x_4 \vee x_2 \vee \neg x_{10}$
$(c_5)\ \neg x_8 \vee x_{10} \vee x_1$ $\quad$ $(c_6)\ x_{10} \vee x_3$
$(c_7)\ x_{10} \vee \neg x_5$
$(c_8)\ x_{17} \vee \neg x_1 \vee \neg x_3 \vee x_5 \vee x_{18}$
$(c_9)\ \neg x_3 \vee \neg x_{19} \vee \neg x_{18}$

$\rho = \{\langle \ldots \neg x_6^1 \ldots \neg x_{17}^1 \rangle \langle (x_8^2) \ldots \neg x_{13}^2 \ldots \rangle \ldots \langle (x_{11}^5) \ldots \rangle\}$.
Le niveau de décision courant est 5 *et $\rho(\mathcal{F}) = faux$.*

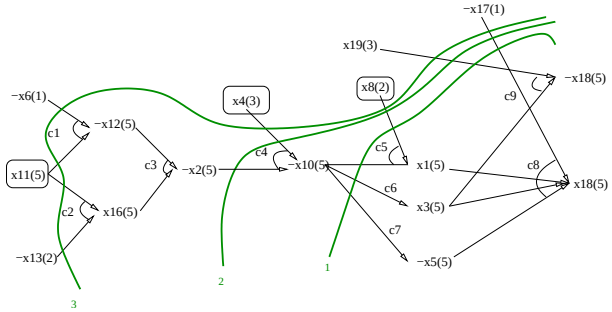


FIG. 1 – Implication Graph $\mathcal{G}_{\mathcal{F}}^{\rho} = (\mathcal{N}, \mathcal{E})$

L'analyse du conflit est le résultat de l'application de la résolution sur la clause devenue fausse (à partir des nœuds conflictuels du graphe d'implications) en utilisant différentes clauses de la forme $(\overline{exp(y)} \vee y)$ implicitement codées à chaque nœud $y \in \mathcal{N}$. On appelle ce processus preuve par résolution basée sur les conflits. En utilisant l'exemple 1, on illuste à présent le schéma d'apprentissage basé sur les conflits (conflict driven clause learning) et ces concepts sous-jacents de clause assertive et unique point d'implication. Le parcours du graphe $\mathcal{G}_{\mathcal{F}}^{\rho}$ permet de générer les clauses suivantes :

- $r_1 = \mathcal{R}[x_{18}, c_8, c_9] = (x_{17}^1 \vee \neg x_1^5 \vee \neg x_3^5 \vee x_5^5 \vee \neg x_{19}^3)$
- $r_2 = \mathcal{R}[x_1, r_1, c_5] = (x_{17}^1 \vee \neg x_3^5 \vee x_5^5 \vee \neg x_{19}^3 \vee \neg x_8^2 \vee x_{10}^5)$
- $r_3 = \mathcal{R}[x_5, r_2, c_7] = (x_{17}^1 \vee \neg x_3^5 \vee \neg x_{19}^3 \vee \neg x_8^2 \vee x_{10}^5)$
- $r_4 = \Delta_1 = \mathcal{R}[x_3, r_3, c_6] = (x_{17}^1 \vee \neg x_{19}^3 \vee \neg x_8^2 \vee x_{10}^5)$

La clause Δ_1 est une *clause assertive* car tout ces littéraux sont affectées à $faux$ avant le niveau du conflit excepté un (x_{10}) qui est affecté au niveau courant 5. À partie de la première clause assertive, on déduit qu'on peut effectué un retour-arrière (backjumping) au niveau 3 (le niveau maximum des littéraux restant dans Δ_1) et affecter x_{10} (appelé littéral assertif) à $vrai$. Le nœud $\neg x_{10}$ dans le graphe (voir coupe 1 dans la Figure. 1) est appelé le premier *Unique Point d'Implication (UIP)*. Toutes les clauses intermidiares r_1, r_2 et r_3 contiennent plus qu'un seul littéral du niveau de décision actuel 5. Si nous continuons ce processus, nous obtenons d'autres clauses assertives $\Delta_2 = (x_{17}^1 \vee \neg x_8^2 \vee \neg x_{19}^3 \vee \neg x_4^3 \vee x_2^5)$ et $\Delta_3 = (x_{17}^1 \vee x_6^1 \vee \neg x_8^2 \vee x_{13}^2 \vee \neg x_4^3 \vee \neg x_{19}^3 \vee \neg x_{11}^5)$ correspon-

dant au second $\neg x_2^5$ (coupe 2 dans la Figure. 1) et au troisième UIP $\neg x_{11}^5$ (coupe 3 dans Figure. 1) respectivement. Le nœud $\neg x_{11}$ est le dernier UIP car il correspond au littéral de décision (voir coupes 2 et 3 dans la Fig. 1).

Notons que pour une clauses assertive $c = (\alpha \vee x)$ déduite par l'analyse de conflit au niveau m, on peut déduire par propagation unitaire x au niveau de décision $i < m$ où $i = l(\alpha)$ i.e., $(\mathcal{F}|_{\rho^i}) \wedge \overline{x} \models_* \bot$. Comme x est affecté à faux au niveau de décision courant m, l'analyse de conflit tend à *corriger* cette affectation en générant une meilleure explication pour le littéral x.

3 Génération de nouvelles explications

Dans cette section, nous montrons que de nouvelles explications pour les littéraux impliqués (littéraux propagés unitairement) peuvent être générées même si une affectation partiel ne mène pas forcèment à un conflit. Ces nouvelles capacités offertes par notre approche sont motivées dans l'exemple ci-dessous.

3.1 Example

Exemple 2 *Soit $\mathcal{F}'$ la nouvelle formule obtenue à partir de $\mathcal{F}$ (voir exemple 1) en substituant la clause c_9 par la nouvelle clause $c_{10} = (x_6 \vee \neg x_{10} \vee x_{18})$. Le graphe d'implications $\mathcal{G}^{\rho}_{\mathcal{F}'}$ est le sous-graphe de $\mathcal{G}^{\rho}_{\mathcal{F}}$ (voir Figure 1) induit par l'ensemble de nœuds $\mathcal{N} \backslash \{\neg x_{18}, x_{19}\}$. Clairement, l'affectation partielle ρ ne mène pas à un conflit au niveau de décision 5. Nous pouvons remarquer que la clause c_{10} n'est pas mémorisée dans le graphe d'implication $\mathcal{G}^{\rho}_{\mathcal{F}'}$ car elle contient deux littéraux $\neg x_{10}$ et x_{18} affectés à $vrai$. En plus, le littéral x_{18} est affecté par propagation unitaire au niveau 5 grâce à la clause c_8 et ρ. Illustrons à présent comment nous pouvons déduire une nouvelle raison (clause assertive) permettant de propager x_{18} à un niveau inférieur à 5. Premièrement, partant de la clause $\overrightarrow{imp}(x_{18})$, nous générons les résolvantes suivante :*

- $r'_1 = \mathcal{R}[x_1, c_8, c_5] = (x_{17}^1 \vee \neg x_8^2 \vee x_{10}^5 \vee \neg x_3^5 \vee x_5^5 \vee x_{18}^5)$
- $r'_2 = \mathcal{R}[x_3, r'_1, c_6] = (x_{17}^1 \vee \neg x_8^2 \vee x_{10}^5 \vee x_5^5 \vee x_{18}^5)$
- $r'_3 = \Delta'_1 = \mathcal{R}[x_5, r'_2, c_7] = (x_{17}^1 \vee \neg x_8^2 \vee x_{10}^5 \vee x_{18}^5)$

La clause Δ'_1 contient seulement deux littéraux x_{10} et x_{18} du niveau de décision actuel 5 et tous les littéraux sont affectés à $faux$ excepté x_{18} qui est affecté à $vrai$. Maintenant, si on applique une étape supplémentaire de résolution entre Δ'_1 et $c_{10} = (x_6 \vee \neg x_{10} \vee x_{18}) \in \mathcal{F}'$, on obtient une nouvelle clause unitaire et assertive $\Delta''_1 = (x_6^1 \vee x_{17}^1 \vee \neg x_8^2 \vee x_{18}^5)$. Cette dernière clause exprime le fait que le littéral x_{18} affecté au niveau 5 peut être propagé au niveau 2. En effet, comme $\mathcal{F}' \models \Delta''_1$ et les deux littéraux x_6 et x_{17} (resp. le littéral $\neg x_8$) sont affectés à $faux$ au niveau 1 (resp. niveau 2), on peut déduire que le littéral x_{18} affecté au niveau 5 peut être affecté par propagation unitaire au niveau 2 grâce à la nouvelle raison Δ''_1.

À partir de l'exemple précédent et contrairement au schéma d'apprentissage classique, on peut effectuer les premières observations suivantes :

- r'_3 n'est pas une clause conflit i.e., $\rho(r'_3) = vrai$
- le littéral x_{18} n'est pas réfuté i.e., la clause déduite Δ''_1 indique que x_{18} doit être affecté à la même valeur de vértié $vrai$ au niveau 2 au lieu du niveau 5

3.2 Présentation Formelle

Donnons à présent une présentation formelle du schéma d'apprentissage de nouvelles raisons. Dans toutes les définitions et propriétés suivantes, on considère $\mathcal{F}$ une formule CNF, ρ une interprétation partielle telle que $(\mathcal{F}|_{\rho})^* \neq \bot$ et m le niveau de décision courant.

Définition 2 (clause bi-assertive) *Une clause $c = (\alpha \vee x \vee y)$ est dite clause bi-assertive ssi $\rho(\alpha) = faux$, $l(\alpha) < m$ et $l(x) = l(y) = m$.*

Une notion similaire de clause bi-assertive est définit par Knot Pipatsrisawat et Adnan Darwiche. Dans [11], une clause bi-assertive est définit comme une clause conflit avec deux littéraux affectés au niveau du conflit alors que dans la définition 2, nous ne faisons aucune hypothèse sur les valeurs de vérité des deux littéraux x et y. Suivant notre définition , la clause r'_3 (exemple 2) satisfaite par x_{18} est une clause bi-assertive. En résumé, et contrairement au travail présenté dans [11] notre définition n'est pas lié à la notion de clause-conflit. Elle peut être générée par preuve bi-assertive (Définition 3) *à chaque nœud* de l'arbre de recherche, même si le grahe d'implications n'est pas un graphe de conflit.

Définition 3 (résolution bi-assertive) *Soit x un littéral tel que $l(x) = m$ et $\overrightarrow{imp}(x)$ est définie. Une résolution bi-assertive $\pi_b(x)$ est une séquence de clauses $\langle r_1, r_2, \ldots r_k \rangle$ satisfaisant les conditions suivantes :*

1. $r_1 = \overrightarrow{imp}(x)$
2. *r_i, pour $i \in 2..k$, est construite en sélectionnant un littéral $y \in r_{i-1}$. La clause r_i est définie donc comme $\mathcal{R}[y, r_{i-1}, \overrightarrow{imp}(\bar{y})]$;*
3. *r_k est une clause bi-assertive*

Comme le littéral $x \in r_1$ ne peut pas être éliminé par résolution, on a $x \in r_k$. En effet, si cette élimination est possible, cela signifie que $\overrightarrow{imp}(x)$ et $\overrightarrow{imp}(\bar{x})$ sont définies. Ceci contredit l'hypothèse $(\mathcal{F}|_{\rho})^* \neq \bot$. De plus, x est l'unique littéral de r_k affecté à $vrai$.

Dans [1], nous avons montré que le littéral assertif x généré en utilisant l'analyse de conflit peut être déduit par propagation unitaire au niveau i, où i est le niveau du saut arrière (ou backjumping) associé à la clause assertive $(\alpha \vee x)$ i.e, $\mathcal{F}|_{\rho^i} \models_* x$. De façon similaire, pour une clause

bi-assertive $(\alpha \vee y \vee x)$, la clause binaire $(y \vee x)$ (voir la propriété 1) peut être aussi déduite par propagation unitaire au niveau $i = l(\alpha)$.

Propriété 1 *Soit $\pi_b(x) = \langle r_1, r_2, \ldots, r_k \rangle$ une résolution bi-assertive telle que $r_k = (\alpha \vee y \vee x)$ où $i = l(\alpha)$. Nous avons $\mathcal{F}|_{\rho^i} \models^* (y \vee x)$*

Preuve 1 *Par définition de la résolution bi-assertive, $\pi_b(x)$ est dérivée en parcourant le graphe d'implications associé à $\mathcal{F}$ et ρ à partir du nœud x. Il est évident que l'affectation $\neg y$ au niveau i mène à la propagation du littéral x. En effet, toutes les clauses utilisées dans la résolution bi-assertive $\pi_b(x)$ contiennent seulement des littéraux de $\bar{\alpha}$ et des littéraux propagés au niveau de décision courant m. Par conséquent, on a $\mathcal{F}|_{\rho^i} \wedge \neg y \models^* x$. Donc $\mathcal{F}|_{\rho^i} \wedge \neg y \wedge \neg x \models^* \bot$.*

Illustrons la propriété 1 en utilisant l'exemple 2. À partir de la clause bi-assertive $r'_3 = (x^1_{17} \vee \neg x^2_8 \vee x^5_{10} \vee x^5_{18})$, il est facile de montrer que la clause $(x^5_{10} \vee x^5_{18})$ peut être déduite par propagation unitaire au niveau 2 i.e, $\mathcal{F}|_{\rho^2} \models^* (x_{10} \vee x_{18})$. A partir de la formule $\mathcal{F}$ et ρ, on peut voir que la formule $\mathcal{F}|_{\rho^2}$ contient les clauses suivantes $\{(x_{10} \vee x_1), (x_{10} \vee x_3), (x_{10} \vee \neg x_5), (\neg x_1 \vee \neg x_3 \vee x_5 \vee x_{18})\}$ (voir le graphe d'implications $\mathcal{G}^{\rho}_{\mathcal{F}'}$). Donc, on a $\mathcal{F}|_{\rho^2} \wedge \neg x_{10} \models^* x_{18}$. Par conséquent, $\mathcal{F}'|_{\rho^2} \wedge \neg x_{10} \wedge \neg x_{18} \models^* \bot$.

Pour dériver une nouvelle raison pour l'implication d'un littéral x donné, on procède en deux étapes. Dans la première, nous générons une résolution bi-assertive $\pi_b(x) = \langle r_1, r_2, \ldots, r_k = (\alpha \vee y \vee x) \rangle$, et dans la seconde, nous éliminons y de r_k par résolution. Ce processus de résolution est appelé résolution assertive, et est formellement défini ci-dessous :

Définition 4 (résolution assertive) *Soit $\pi_u(x)$ une séquence de clauses $\langle r_1, \ldots, r_{k-1}, r_k \rangle$. $\pi_u(x)$ est une résolution assertive si elle satisfait les deux conditions suivantes :*

1. *$\langle r_1, \ldots, r_{k-1} = (\alpha \vee y \vee x) \rangle$ est une résolution bi-assertive*
2. *$\exists c = (\beta \vee \neg y \vee x) \in \mathcal{F}$ une clause bi-assertive $r_k = \mathcal{R}[y, r_{k-1}, c] = (\gamma \vee x)$ où $\gamma = \alpha \cup \beta$.*

Il suit de la définition 2 que tous les littéraux de α et β sont affectés à $faux$ avant le niveau m. Par conséquent, la résolvante r_k est une clause assertive unitaire. Cela signifie que lorsque $\pi_u(x)$ existe, on déduit que le littéral x propagé au niveau m aurait dû être propagé à $vrai$ au niveau $k < m$ où $k = l(\gamma)$. La propriété suivante fait état de ce résultat.

Propriété 2 *Soit x un littéral tel que $l(x) = m$ et $\overrightarrow{imp}(x)$ existe. Si $\pi_u(x) = \langle r_1, \ldots, r_{k-1}, r_k = (\gamma \vee x) \rangle$ est une résolution assertive alors $\mathcal{F}|_{\rho^i} \models^* x$ où $i = l(\gamma)$.*

Algorithm 1: algorithme RDBA

```
   Données: A CNF formula F;
   Résultat: vrai if F is satisfiable ; faux otherwise
1  begin
2      currentLevel = 0, Δ = ∅;
3      while (vrau) do
4          if (propagate()==faux) then
5              if (currentLevel==0) then return faux;
6              c=(α ∨ ¬d)=analyze();
7              Δ = Δ ∪ c;
8              backJumpLevel = l(α);
9              bactrack(backJumpLevel+1);
10             (newJumpLevel, Δ′) =
               learnForNewReasons(backJumpLevel+1);
11             Δ = Δ ∪ Δ′;
12             bactrack(newJumpLevel);
13         else
14             currentLevel++;
15             if (decide()== faux) then
16                 return vrai
17 end
```

Preuve 2 *Premièrement, comme $\langle r_1, \ldots, r_{k-1} \rangle$ est une résolution bi-assertive, la résolvante r_{k-1} est de la forme $(\alpha \vee y \vee x)$ où $l(\alpha) \leq i$. En utilisant la propriété 1, on peut déduire que $\mathcal{F}|_{\rho^i} \models^* (y \vee x)$, alors $\mathcal{F}|_{\rho^i} \wedge \neg x \models^* y$. De plus, par définition de $\pi_u(x)$, la résolvante r_k est obtenue par résolution sur y entre r_{k-1} et $c \in \mathcal{F}$ de la forme $(\beta \vee \neg y \vee x)$ où $l(\beta) \leq i$. Comme $\forall z \in \beta\ \rho(z) = faux$, donc $(\neg y \vee x) \in \mathcal{F}|_{\rho^i}$ et $\mathcal{F}|_{\rho^i} \wedge \neg x \models^* \neg y$. Par conséquent, $\mathcal{F}|_{\rho^i} \wedge \neg x \models^* \bot$.*

Dans la propriété 2, nous avons montré que le littéral x peut être déduit par propagation unitaire au niveau i. Cependant, vérifier si chaque littéral non affecté peut être déduit par propagation unitaire à chaque nœud de l'arbre de recherche est coûteux. Dans la section suivante, on montre comment quelques unes de ces déductions peuvent être obtenues à la volée en utilisant notre approche d'apprentissage de nouvelles raisons.

4 Réordonnancement dynamique à base d'apprentissage

Nous présentons dans cette section comment le schéma d'apprentissage de nouvelles raisons peut être utilisé pour réordonner dynamiquement les affectations partielles d'un solveur SAT moderne. L'amélioration du solveur SAT est esquissée dans l'algorithme 1. Pour des raisons de simplicité, l'algorithme ne contient pas l'une des composante essentiel des solveurs SAT à savoir le redémarrage. Nous rappelons néanmoins les principales fonctions qui y sont incorporées :

- *propagate* : effectue la propagation unitaire et retourne $faux$ en cas de conflit et $vrai$ sinon.
- *analyze* : retourne la clause apprise c calculée en utilisant le schéma classique d'apprentissage. Le littéral

assertif est dénoté d.

- *learn* : ajoute une clause apprise c à la base des clauses apprises
- *learnFromSuccess* : applique notre schéma d'apprentissage de nouvelles raisons présenté dans l'algorithme 2
- *backtrack* : effectue un retour-arrière au niveau passé en paramètre.
- *decide* : sélectionne et affecte le littéral de décision suivant. La fonction retourne $vrai$, si la formule est satisfaite (tous les littéraux sont affectés), et $faux$ sinon.

Comme on l'a vu dans la section précédente, l'apprentissage de nouvelles raisons peut être appliqué à chaque nœud de l'arbre de recherche. En d'autres termes, pour chaque littéral unitaire y propagé à un certain niveau i, on peut appliquer le schéma d'apprentissage de nouvelles raisons pour déduire une implication à un niveau précédent i. Cependant, une application systématique peut dégrader les performances des solveurs. Dans l'algorithme 1, on applique l'apprentissage de nouvelles raisons (ligne 10) seulement lorsqu'un conflit est détecté i.e, si l'appel à *propagate()* retourne $faux$. Plus précisément, on utilise le composant classique d'analyse de conflit (ligne 6). La clause assertive c générée est ajoutée à la base des clauses apprises Δ (ligne 7). L'algorithme effectue donc un retour-arrière au niveau $backJumpLevel+1$ (ligne 9) et l'apprentissage de nouvelles raisons est seulement appliqué à ce niveau particulier. En restreignant ce schéma d'apprentissage aux littéraux y propagés au niveau $backJumpLevel + 1$, on garantit que chaque nouveau niveau d'implication i pour x est inférieur ou égal au niveau de retour-arrière courant. Un ensemble de nouvelles raisons Δ' et de nouveaux niveaux (appelé $newJumpLevel$) est retourné par la fonction $learnForNewReasons$ (ligne 10). le nouveau niveau de retour-arrière est calculé en choisissant le meilleur niveau (le plus petit) de ces nouvelles raisons Δ'. Avant d'effectuer un retour-arrière à ce nouveau niveau (ligne 12), l'ensemble de ces nouvelles raisons Δ' est ajouté à la base des clauses apprises (ligne 11). Si la procédure *learnForNewReasons* ne trouve pas de nouvelles implications, $newJumpLevel$ est égale à $backJumpLevel$, et l'algorithme suit le schéma classique d'apprentissage CDCL.

La fonction $learnFromNewReasons$ est détaillée dans l'algorithme 2. Premièrement, y (resp. U) désigne le littéral de décision (resp. l'ensemble des littéraux propagés à ce niveau) affecté au niveau $currentLevel$ (initialisé à $backJumpLevel + 1$) (ligne 2 et ligne 3). Pour chaque littéral $w \in U$, si on dispose d'une clause bi-assertive de la forme $c = (\beta \vee y \vee w) \in \mathcal{F}$ telle que $\rho(\beta) = faux$, $\rho(w) = \rho(y) = vrai$ est détecté, alors une résolution assertive $\pi_u(w) = \langle r_1, \ldots, r_k \rangle$ est opérée (voir ligne 7). La nouvelle implication r_k de w est ajoutée à la base des clauses apprises (ligne 8) et le $newJumpLevel$ est mis à jour (ligne 9). Tandis que les littéraux propagés à ce niveau $backJumpLevel + 1$ sont traités, le meilleur niveau d'implication ($newJumpLevel$) est retourné (ligne 10), et l'algorithme 1 peut effectuer un retour-arrière à ce nouveau niveau (ligne 12).

Algorithm 2: learnForNewReasons

Données: $currentLevel$: application level of learning for new reasons
Résultat: $newJumpLevel$: the highest implication level found; R : a set of new reasons;

```
1  begin
2      y = decisionLiteral(currentLevel);
3      U=UPLiterals(currentLevel);
4      newJumpLevel = currentLevel-1, R = ∅;
5      for (w ∈ U) do
6          if (∃c = (β ∨ y ∨ w) ∈ F s.t. ρ(β) = faux,
           ρ(w) = ρ(y) = vrai) then
7              Soit π_u(w) = ⟨r_1, ..., r_{k-1} = (α ∨ ¬y ∨ w), r_k⟩
           où r_k = (γ ∨ w) = R[y, r_{k-1}, c] tq. γ = α ∪ β;
8          R = R ∪ r_k;
9          newJumpLevel = min(l(γ),newJumpLevel);
10     return (newJumpLevel, R);
11 end
```

Pour réarranger l'affectation partielle, on utilise le système de sauvegarde similaire à celui proposé dans [10]. En effet, durant le retour-arrière, tous les littéraux de décisions entre $backJumpLevel$ (calculé à partir de l'apprentissage classique) et le $newJumpLevel$ (calculé par $learnForNewReasons$) sont enregistrés. La fonction $decide()$ les sélectionne en priorité dans le but de maintenir approximativement les mêmes décisions. Les littéraux unitaires avec les nouvelles implications sont affectés au bon niveau (le nouveau niveau calculé) grâce aux raisons enregistrées (voir ligne 12 dans l'algorithme 2). Ce processus définit notre approche de réordonnancement dynamique basé sur l'apprentissage (RDBA).

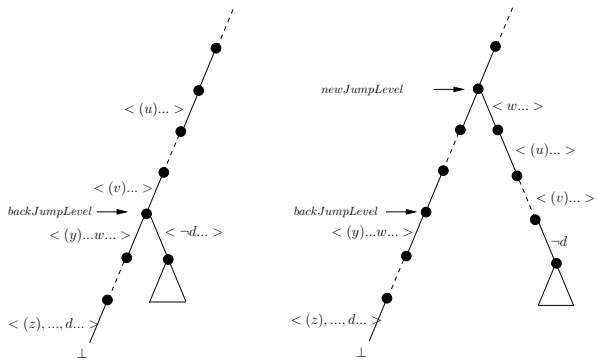


FIG. 2 – CDCL Vs RDBA

La figure 2 illustre la différence entre un branchement CDCL (à gauche) et RDBA (à droite). Soit $(\alpha \vee \neg d)$ la clause assertive obtenue par l'analyse de conflit classique. Dans l'approche CDCL, l'algorithme effectue un retour-

arrière au niveau $backJumpLevel = l(\alpha)$ et affecte le littéral assertif $\neg d$. Dans l'approche RDBA, l'algorithme effectue un retour-arrière au niveau $backJumpLevel + 1$ et applique le processus d'apprentissage de nouvelles raisons sur tous les littéraux propagés à ce niveau (littéral w dans la figure). Toutes les nouvelles implications pour ces littéraux sont enregistrées dans la base des clauses apprises. Supposons que le meilleur niveau de retour-arrière ($newJumpLevel$) correspond à la nouvelle implication pour w. L'algorithme effectue un retour-arrière au niveau $newJumpLevel$ et affecte le littéral w et les autres littéraux grâce à la mémorisation de l'implication. Finalement, la recherche continue en affectant en priorité les littéraux de décision entre $backJumpLevel$ et $newJumpLevel$ seulement et dans l'ordre tant que c'est possible.

5 Expérimentations

Nous avons effectués des expérimentations sur un large panel de problèmes industriels issus des deux dernières compétitions SAT-Race'06 et SAT'2007. Avant les tests de résolution toutes les instances sont pré-traiter par *SatELite* [3]. Le temps limite de calcul est fixé à 1800 secondes et les résultats sont indiqués en secondes. Nous avons intégré l'extension RDBA (section 4) dans Minisat et nous avons effectué une comparaison entre Minisat classique et Minisat incluant cette approche. Ces tests ont été menés sur un cluster Xeon 3.2GHz (2 GB RAM).

Les représentations en echelle logarithmique dans les figures 3 et 4 illustrent une comparaison des résultats de Minisat et Minisat+RDBA sur des instances satisfiables et insatisfiables respectivement. L'axe des x (resp. y) correspond à Minisat+RDBA (resp. Minisat). Chaque instance est associée à un point de coordonnées (tx, ty) qui représente les temps de calcul obtenus par les deux approches sur l'instance donnée.

La figure 3 montre que sur les instances satisfiables, l'apprentissage de nouvelles raisons permet d'accélérer le processus de résolution par rapport à Minisat. La majorité des points de la figure sont situés au-dessus de la diagonale i.e., Minisat+RDBA est meilleur. Ces résultats ne sont pas surprenants car notre approche vise à corriger le niveau et non pas la valeur de vérité des littéraux impliqués. Ce qui est diffèrent de l'analyse de conflit classique, où on modifie à la fois le niveau d'affectation (niveau assertif) et la valeur de vérité du littéral assertif. D'un autre côté, pour les instances insatisfiables (voir figure 4), il n'y a pas de séparation claire entre les performances des deux approches.

Les figures 3 et 4 (figures du bas) montrent le $temps$ mis par chaque solveur pour résoudre un certain nombre d'instances. Cette vision globale confirme que notre approche est meilleure sur les instances satisfiables en terme de temps de calcul. Au contraire sur les instances insatisfiables, la différence entre les performances est moins

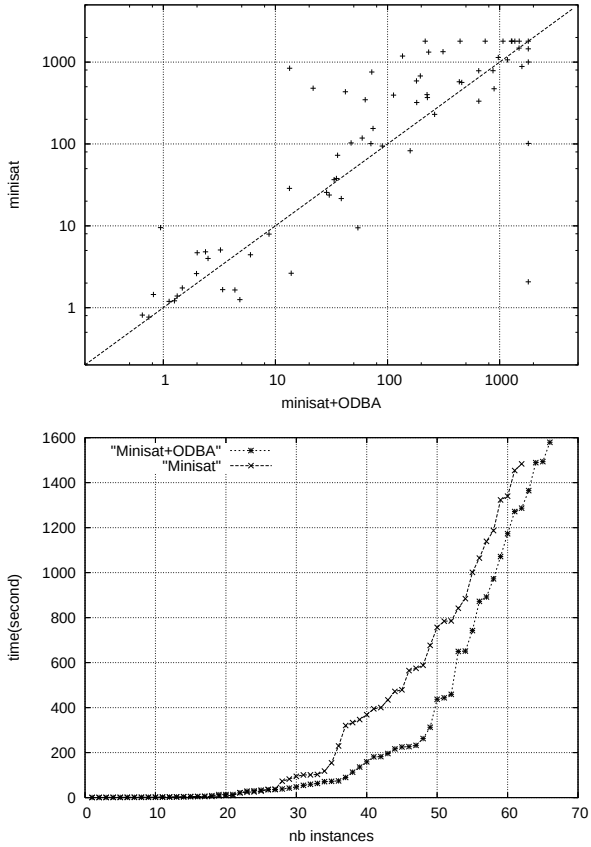


FIG. 3 – Satisfiables instances

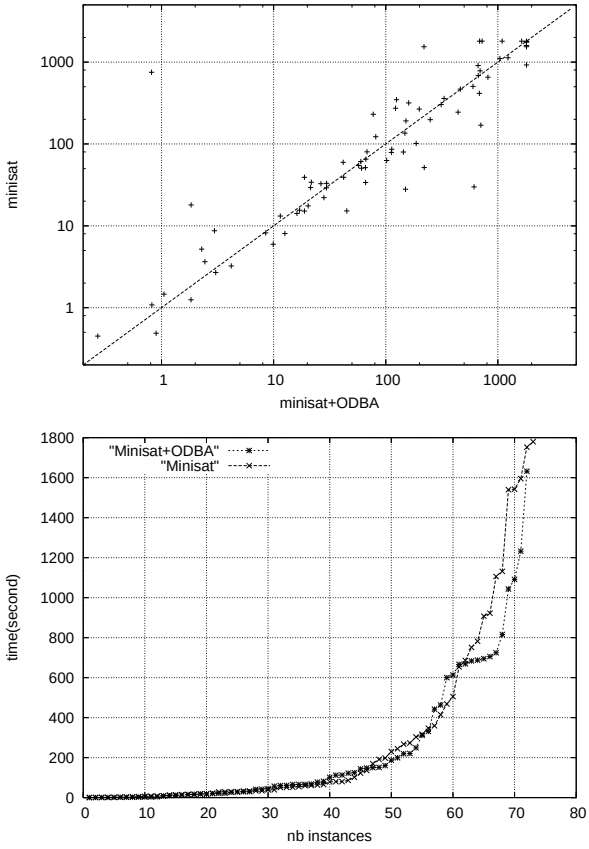


FIG. 4 – Unsatisfiables instances

flagrante mais en moyenne Minisat+RDBA est légèrement meilleur.

Finalement, la table 1 résume quelques résultats obtenus par Minisat+RDBA et Minisat. Pour chaque famille (première colonne), on indique le nombre d'instances (deuxième colonne), la moyenne du temps de calcul sur les instances résolues et le nombre d'instances résolues (entre parenthèses). Les résultats sont donnés sur des classes (SAT) et (UNSAT) et sur toutes les instances (SAT-UNSAT). Pour chaque famille, les meilleurs résultats en terme de temps moyen ou de nombre d'instances résolues sont mises en gras pour plus de lisibilité. Comme on peut le remarquer sur ces résultats Minisat+RDBA obtient de meilleurs résultats en général sur des familles satisfiables. Sur la famille $safe-*$ par exemple, RDBA permet un gain de près de deux ordres de grandeur.

Pour résumer, l'intégration de RDBA sur un solveur moderne est assez prometteuse car elle a mis en lumière une question principale sur l'intérêt de l'ordre d'affectation dans les solveurs modernes. Cette approche est complémentaire au schéma classique d'apprentissage dans la mesure où elle offre une alternative, qui, combinée à CDCL, permet comme cela a été dit de corriger l'interprétation courante.

6 Conclusion

Dans cet article, un nouveau cadre d'apprentissage de nouvelles raisons d'implication est proposé. Ce nouveau cadre permet de dériver de meilleures implications pour la propagation unitaire des littéraux. Ceci peut être vu comme une méthode de réarrangement de l'interprétation courante. Alors que dans les solveurs SAT la partie satisfaite de la formule est ignorée, notre approche suggère que ces clauses connectées au reste de la formule peuvent être exploitées avantageusement durant la recherche. Notre première intégration de RDBA dans un solveur moderne (Minisat) montre une nette amélioration du temps de calcul sur certaines classes de problèmes industriels. Plus intéressant encore, nous avons constaté une relation entre la résolution assertive et la déduction basée sur la propagation unitaire. Comme notre approche essaye de réarranger l'interprétation courante, nous envisageons d'approfondir la question du rôle joué par les redémarrages dans ce contexte. Finalement, étendre ce schéma pour récupérer d'autres formes de consistance plus fortes que la propagation unitaire nous semble être une piste de recherche pouvant permettre la résolution de problèmes difficiles.

Références

[1] G. Audemard, L. Bordeaux, Y. Hamadi, S. Jabbour, and L. Sais. Generalized framework for conflict analysis. In *In proceedings of eleventh International*

		SAT		UNSAT		SAT-UNSAT	
familles	# inst.	Minisat+RDBA	Minisat	Minisat+RDBA	Minisat	Minisat+RDBA	Minisat
mizh_*	10	**501**(10)	720(10)	–	–	**501**(10)	720(10)
manol_*	20	–	–	**336(17)**	344(14)	**336(17)**	344(14)
partial_*	20	**1272(1)**	–	–	–	**1271(1)**	–
total_*	20	262**(7)**	94(4)	66(3)	66(3)	203**(10)**	82(7)
vmp_grieu_*	12	**462**(4)	801(4)	–	–	**462(4)**	801(4)
APro_*	16	13(1)	**2**(1)	323(7)	557**(9)**	284(8)	502**(10)**
dated_*	20	160(9)	**151**(9)	135(2)	661(3)	156(11)	279**(12)**
clause_*	4	**233**(4)	303(4)	–	–	**233**(4)	303(4)
cube_*	4	891(1)	**472**(1)	**59**(1)	60(1)	475(2)	**266**(2)
gold_*	7	–	–	597(4)	**577**(4)	597(4)	**577**(4)
safe_*	4	**13**(1)	841(1)	–	–	**13(1)**	841(1)
ibm_*	20	**98**(10)	121(10)	**36**(9)	102(9)	**69**(19)	112(19)
IBM_*	35	1295**(4)**	1113(3)	**331**(1)	359(1)	1102**(5)**	924(4)
simon_*	6	**149**(2)	277(2)	204(3)	**128**(3)	**182**(5)	188(5)
block_*	2	–	–	27(2)	27(2)	27(2)	27(2)
dspam_*	2	–	–	315(2)	**34**(2)	315(2)	**34**(2)
schup_*	2	**71**(1)	100(1)	**666**(1)	907(1)	**368**(2)	504(2)
sort_*	5	**312**(1)	1339(1)	1232(1)	**1131**(1)	**772**(2)	1235(2)
velev_*	12	–	**2(1)**	25(3)	**19**(3)	25(3)	**15(4)**

TAB. 1 – Highlighted results

Conference on Theory and Applications of Satisfiability Testing (SAT'2008), 2008.

[2] M. Davis, G. Logemann, and D. W. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7) :394–397, 1962.

[3] N. Eén and A. Biere. Effective preprocessing in SAT through variable and clause elimination. In *Proceedings of the Eighth International Conference on Theory and Applications of Satisfiability Testing (SAT'05)*, pages 61–75, 2005.

[4] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In *Proceedings of the Sixth International Conference on Theory and Applications of Satisfiability Testing (SAT'03)*, pages 502–518, 2002.

[5] C. P. Gomes, B. Selman, N. Crato, and H. Kautz. Heavy-tail phenomena in satisfiability and constraint satisfaction. *Journal of Automated Reasoning*, pages 67 – 100, 2000.

[6] Carla P. Gomes, Bart Selman, and Henry Kautz. Boosting combinatorial search through randomization. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI'97)*, pages 431–437, Madison, Wisconsin, 1998.

[7] H. Kautz, E. Horvitz, Y. Ruan, C. Gomes, and B. Selman. Dynamic restart policies. In *AAAI'02*, pages 674–682, 2002.

[8] Joao P. Marques-Silva and Karem A. Sakallah. GRASP - A New Search Algorithm for Satisfiability. In *Proceedings of IEEE/ACM International Conference on Computer-Aided Design*, pages 220–227, November 1996.

[9] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff : Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference (DAC'01)*, pages 530–535, 2001.

[10] Knot Pipatsrisawat and Adnan Darwiche. A lightweight component caching scheme for satisfiability solvers. In *Proceedings of 10th International Conference on Theory and Applications of Satisfiability Testing(SAT)*, pages 294–299, 2007.

[11] Knot Pipatsrisawat and Adnan Darwiche. A new clause learning scheme for efficient unsatisfiability proofs. In *AAAI'2008*, pages 1481–1484, 2008.

CSP dynamiques pour la génération de tests de systèmes réactifs

Christophe Junke, Benjamin Blanc

CEA LIST, Laboratoire de Sûreté des Logiciels
Boîte 65, F-91191 Gif-sur-Yvette France
prenom.nom@cea.fr

Résumé

GATeL est un outil de génération de tests dédié à la validation et la vérification de programmes réactifs synchrones. À partir d'une spécification décrivant un système réactif, de propriétés de son environnement ainsi que d'un objectif de test décrits en `Lustre`, les entrées successives du système menant à la réalisation de l'objectif de test sont générées. GATeL repose sur une interprétation en programmation logique par contraintes de `Lustre`, à l'aide de contraintes booléennes, arithmétiques et temporelles. Nous présentons dans cet article la propagation dynamique de contraintes avec stratégie en arrière de GATeL, qui construit de manière paresseuse les éléments du passé nécessaires à la réalisation d'un objectif de test.

1 Introduction

Le test logiciel [12] consiste à exécuter un logiciel dans le but de trouver ses défauts. C'est une activité dont l'automatisation pose plusieurs difficultés : sélectionner des cas de tests, trouver les entrées correspondantes et déterminer le succès ou l'échec à l'issue de l'exécution du test (oracle). Plusieurs outils proposent une approche automatique à base de programmation par contraintes pour les deux derniers points (INKA [9], BZ-TT [4], PathCrawler [16],Osmose [2]). À partir d'un problème de génération composé d'un programme et d'un objectif de test, ces outils produisent un problème de satisfaction de contraintes (CSP[1]) dans lequel les variables représentent les entrées, les domaines identifient les types de données, et les contraintes expriment les opérateurs du langage traité. Généralement, les langages de programmation permettent d'écrire des systèmes qui se traduisent par des CSP n'appartenant pas à une classe bien identifiée de problèmes de résolution. De plus, les domaines représentant les types numériques sont très grands et ne permettent pas de réutiliser les techniques mises au point pour des domaines finis. Les outils doivent alors combiner des solveurs spécifiques, et ce à un niveau suffisamment haut pour permettre le maximum de déductions entre les contraintes [10]. C'est pourquoi les équipes citées ont développé soit un solveur complet, soit une couche permettant de contrôler l'appel à des solveurs spécifiques.

GATeL. Nous présentons dans cet article l'interprétation en CSP du problème de génération de tests utilisée dans GATeL [11], un outil de validation et de vérification pour les systèmes réactifs. Un *système réactif* est un programme qui permet d'observer et de contrôler un environnement extérieur, physique ou logiciel, en interagissant avec lui de manière continue. L'approche flots de données synchrones [3] est un paradigme reconnu pour la modélisation de systèmes réactifs critiques : les langages synchrones disposent d'une sémantique formelle dans laquelle les calculs à chaque cycle sont déterministes en fonction des entrées successives du système. Cette approche facilite entre autres le développement de compilateurs certifiés, ce qui permet aux industriels de ces domaines, qui sont soumis à des normes strictes de validation et de vérification, de simplifier leurs processus de vérification. Nous nous intéressons au langage `Lustre` [5] ainsi qu'à son équivalent industriel, `Scade`[2]. Il existe un compilateur certifié (DO-178B) pour `Scade`, mais son utilisation n'est pas suffisante pour supprimer les

[1] Constraint Satisfaction Problem

[2] `http://www.esterel-technologies.com`

phases de tests car les normes imposent le test exhaustif des comportements décrits par la spécification (test fonctionnel). Pour chacun de ces comportement, GATeL peut générer des séquences d'entrées amenant le système de l'état initial jusqu'à la situation voulue.

Approche incrémentale. Pour les langages usuels (C, Java, B, etc.), une difficulté dans la génération des entrées est due à la présence d'itérateurs. Bien que ce type d'opérateur n'existe pas en `Lustre`, le langage possède implicitement une structure de contrôle itérative. En effet, il manipule des flots de données synchrones, qui peuvent être représentés à tout instant par des *séquences finies* de valeurs issues d'un état initial. Chaque flot est défini de manière mutuellement récursive par rapport à ses valeurs passées, et à celles d'autres flots. La résolution de contraintes correspond alors à une exploration synchronisée des définitions récursives de chaque flot apparaissant dans le problème.

GATeL propose une interprétation en Programmation Logique par Contraintes (PLC) de ce problème de génération de tests, ainsi qu'un solveur spécifique traitant les expressions de façon paresseuse, selon une stratégie de génération *en arrière*.À partir d'un CSP initial correspondant à l'interprétation de l'objectif de test, GATeL explore les passés possibles aboutissant à cet objectif. Une approche visant à construire des CSP statiques pour chaque longueur de séquence serait notoirement inefficace, dans la mesure où l'on se retrouverait dans un schéma *generate-and-test*. C'est pourquoi, afin de limiter la taille des séquences produites, l'exploration de nouveaux cycles est faite incrémentalement. Elle exploite les expressions temporelles relatives au contrôle du système sous test, qui contraignent les chemins d'exécution et le nombre de cycles nécessaires à l'élaboration d'un cas de test : la création de nouveaux cycles dans le passé ne se fait que lorsque le CSP courant l'impose. Cette exploration dynamique du passé est généralisée à celle des expressions booléennes, dont dépend aussi le contrôle, en les interprétant de manière paresseuse. Les expressions numériques sont quant à elles déléguées de manière classique à un solveur sur les domaines finis ou continus.

Plan. Nous présentons dans la section 2 les constructions du langage `Lustre`. La section 3 rappelle des définitions concernant les CSP dynamiques et donne les bases de notre interprétation. Nous montrons ensuite comment les contraintes de différentes natures sont construites efficacement (sec. 4). Des heuristiques de résolution appropriées, que nous abordons dans la partie 6, permettent de traiter des problèmes de génération de tests nécessitant des séquences très longues (en milliers de cycles). Nous présentons enfin dans la section 7 des extraits d'études de cas pour lesquels les performances obtenues sont mises en évidence.

2 Lustre

`Lustre` est un langage de modélisation utilisé principalement pour décrire des programmes réactifs de contrôle/commande. À chaque itération de la boucle réactive , les capteurs fournissent au système des valeurs d'entrées, qui servent à calculer les sorties destinées aux actionneurs. Les valeurs successives des entrées et sorties au cours du temps constituent des flots de données. Ainsi, toutes les expressions du langage dénotent une suite infinie de valeurs dans le temps : la constante `3` désigne le flot $(3, 3, \dots)$, l'expression `A+B` le flot $(A_0 + B_0, A_1 + B_1, \dots)$.

Approche synchrone. `Lustre` est un modèle de flots de données *synchrone* car l'ensemble de ces flots doivent avoir la même longueur à tout instant. Pour garantir cette propriété du modèle dans une application réelle, il est nécessaire de pouvoir borner le temps de calcul d'une itération de boucle pour l'ensemble des sorties du programme, afin d'obtenir le temps de cycle minimum du contrôleur. Les constructions présentes dans le langage assurent que cette borne existe pour chaque modèle `Lustre`.

Pour décrire des réactions temporelles complexes, `Lustre` permet de faire référence à des valeurs passées des flots de données, à travers l'opérateur de retard unitaire « `pre` ». C'est un opérateur unaire qui s'applique à un tout flot `X` de la manière suivante : `pre(X)` = $(\text{NIL}, X_0, X_1, X_2, \dots)$. L'opérateur `pre` réalise un décalage temporel d'un cycle. Puisqu'il n'existe pas de cycle antérieur au cycle initial, cet opérateur n'est pas défini au cycle 0, ce qui est figuré par le terme NIL dans l'exemple. Par extension, toute opération qui serait réalisée au cycle initial sur le flot `pre(X)` aurait un résultat indéterminé. Ce problème est levé par la deuxième primitive temporelle de `Lustre`, notée « `->` ». Cet opérateur binaire permet de définir la valeur initiale d'un flot comme suit : soient `A` et `B` deux flots `Lustre` de même type, alors `A -> B` = $(A_0, B_1, B_2, B_3, \dots)$.

Syntaxe du noyau `Lustre`. Un modèle `Lustre` est organisé en un ou plusieurs nœuds, définissant chacun un système d'équations où les flots de sortie sont exprimés en fonction de flots d'entrée par l'intermédiaire d'expressions arithmétiques, logiques ou temporelles. La figure 1 décrit la syntaxe abstraite d'un noyau de `Lustre`. Bien que celles-ci soient prises en

$$
\begin{array}{rcl}
\textit{expr} & ::= & \textit{bool} \mid \textit{integer} \mid \textsc{id} \\
& \mid & \texttt{pre}\ \textit{expr} \\
& \mid & \textit{expr}\ \texttt{->}\ \textit{expr} \\
& \mid & \textit{expr}\ \textsc{cmp}\ \textit{expr} \\
\textit{bool} & ::= & \textit{bool}\ \texttt{and}\ \textit{bool} \\
& \mid & \textit{bool}\ \texttt{or}\ \textit{bool} \\
& \mid & \texttt{not}\ \textit{bool} \\
& \mid & \texttt{true} \mid \texttt{false} \\
\textit{integer} & ::= & \textit{integer}\ \textsc{op}\ \textit{integer} \\
& \mid & \texttt{-}\ \textit{integer} \\
& \mid & \textsc{int} \\
\textsc{cmp} & ::= & = \mid <> \mid \leq \mid \geq \mid < \mid > \\
\textsc{op} & ::= & + \mid - \mid * \mid \textit{div} \mid \textit{mod}
\end{array}
$$

FIG. 1 – Syntaxe abstraite.

compte dans GATeL, nous ne nous intéressons pas ici aux constructions utilisant la notion d'horloge synchrone. De plus, nous ne traitons pas le cas des données de type réel dans cet article.

Pour permettre la déclaration d'objectifs de test, GATeL étend le langage par une directive spécifique, notée `reach`. Cette directive prend en argument n'importe quelle expression booléenne `Lustre` : l'expression `reach(Exp)` signifie que l'on souhaite construire des séquences pour lesquelles la condition `Exp` est vérifiée au dernier cycle de la séquence. Cette expression ayant la même expressivité que le langage, elle peut correspondre à la détection d'une alarme, ou encore décrire un scénario de test faisant transiter le système par des états particuliers au cours du temps.

```
node TEMPO(start,abort:bool;tempo:int(*!0..50!*))
returns (end_tempo: bool);
var counter, sel_tempo: int ;
    set: bool;
let
 set = start and not(abort) ->
       ( if   pre(set)
         then not(abort)
         else start and not(abort) );

 sel_tempo = if   start and not(abort)
             then tempo
             else 0 -> pre(sel_tempo);

 counter = if   set and not(start)
           then 1 -> pre(counter) + 1
           else 0;

 end_tempo = (counter > sel_tempo);
tel;
```

FIG. 2 – Le nœud `TEMPO`.

Exemple. La figure 2 montre la spécification d'une temporisation, représenté par le nœud `TEMPO`. Le système dispose de trois entrées et d'une sortie : lorsque la temporisation est démarée à l'aide de `start`, un compteur interne, `counter`, est incrémenté à chaque cycle d'une unité, jusqu'à ce qu'il atteigne ou dépasse la durée mémorisée au lancement du décompte, `sel_tempo`. La variable booléenne `end_tempo` prend alors la valeur `true`. À tout moment, la variable `abort` peut réinitialiser le compteur et arrêter le décompte. La figure 3 montre une trace d'exécution du nœud `TEMPO` illustrant le fonctionnement de la variable `set`. Celle-ci représente l'état de la temporisation, qui peut être active ou inactive. Elle vaut *true* à partir du moment où `start` passe à *true*, et reste dans cette état jusqu'à ce que `abort` devienne vrai.

Cycles	0	1	2	3	...
start	*false*	*true*	*false*	*false*	
abort	*false*	*false*	*false*	*true*	
E1	*true*	*true*	*true*	*false*	
E2	*false*	*true*	*false*	*false*	
E3		*false*	*true*	*true*	
E4		*true*	*true*	*false*	
set	*false*	*true*	*true*	*false*	

```
E1  = not(abort)
E2  = start and E1
E3  = pre(set)
E4  = if E3 then E1 else E2
set = E2 -> E4
```

FIG. 3 – Chronogramme d'exécution de `set`.

3 Construction dynamique du CSP

La génération de séquences de tests est un problème pour lequel le nombre d'itérations de la boucle réactive principale, nécessaire à la réalisation d'un objectif de test, n'est pas connu à l'avance. De plus, et bien qu'une analyse statique du programme sous test soit effectuée avant la génération de tests, il existe potentiellement des chemins de calculs très longs n'intervenant pas dans la réalisation d'un objectif de test donné. Afin de ne travailler que sur des contraintes liées au problème de génération, nous choisissons d'introduire ces contraintes de manière paresseuse, ce qui passe par la manipulation d'un CSP dynamique. Nous décrivons ici quelle interprétation de la notion de CSP dynamique nous partageons parmi celles qui ont été proposées dans la littérature. Tout d'abord, nous rappelons celle d'un CSP statique.

Définition 3.a. *CSP statique*

Un CSP statique P est donné par un triplet (V, D, C) représentant des ensembles de variables, de domaines et de contraintes.
À chaque variable v_i de l'ensemble $V = \{v_1, \ldots, v_n\}$, est associé un domaine D_i par une relation $dom(v_i) = D_i$.
Une contrainte de C est un couple $\langle R, \Sigma \rangle$ où R est une relation restreignant les domaines de valeurs des variables de l'ensemble $\Sigma \subseteq V$.
On appelle contrainte *atomique* $C = \langle R, \Sigma \rangle$ une contrainte pour laquelle l'ensemble des variables Σ est de cardinalité fixe. Par opposition, une contrainte est dite *globale* si la cardinalité de Σ n'est pas connue.

Il y a deux manières de considérer les CSP dynamiques, selon que les modifications sont faites depuis l'intérieur ou l'exterieur de la procédure de recherche de solutions [15]. L'évolution est externe lorsque le caractère dynamique reflète des modifications d'un problème selon son environnement ou une interaction particulière. Par exemple, les problèmes de gestion de ressources sont dynamiques car les données du problèmes peuvent changer au cours du temps (changement d'une activité dans un planning, retrait ou ajout d'une ressource, ...). Dans ce cas, un problème intéressant lié à ces problèmes dynamiques consiste à garantir la stabilité des solutions tout en étant efficace, et cela passe par la réutilisation de solutions d'une modification à une autre du problème [14]. Il existe ensuite deux types d'évolution interne de CSP dynamiques. Un premier type consiste à ne considérer qu'un certain nombre de variables du CSP à un instant donné en fonction de l'instanciation partielle [13]. Le second type d'évolution interne apparait lorsque le CSP manipule des contraintes globales pouvant elles-mêmes introduire d'autres contraintes (atomiques ou globales). Le moteur d'INKA traduit par exemple les boucles *while* par une telle contrainte, car la valeur de vérité de la boucle est une variable du CSP, et son instanciation peut introduire de nouvelles variables et contraintes correspondant au corps de sa boucle. C'est ce type d'évolution que nous mettons en oeuvre dans GATeL, pour gérer l'aspect temporel et logique du flot de contrôle.

Définition 3.b. *CSP dynamique*

Si $P = (V, D, C)$ est un CSP, alors $P' = (V', D', C')$ tel que $V \subseteq V'$, $D'_v \subseteq D_v$ $(\forall v \in V)$, et $C \subseteq C'$ est une *restriction* de P.
Un CSP dynamique est une succession de problèmes P_i telle que chaque P_i soit une restriction de P_{i-1}.

Ainsi, un problème de satisfaction de contraintes est dynamique lorsqu'il existe une suite de transitions entre des CSP statiques [7]. Nous nous intéressons ici uniquement aux transitions provoquant des restrictions successives. Dans la suite de cette section, nous décrivons ce que représentent les ensembles (V, D, C) dans le cadre de notre problème de génération de tests.

Variables. Nous introduisons une matrice $\mathcal{M}$ de variables logiques, où chaque ligne représente un flot nommé du modèle `Lustre` (entrée ou sortie), et le nombre de colonnes est variable et correspond au nombre de cycles nécessaires pour atteindre l'objectif. Le point essentiel de notre interprétation est d'introduire les variables logiques dynamiquement, en fonction des besoins. Afin de permettre une résolution du problème de contraintes de façon efficace, toutes les variables d'une colonne de la matrice ne sont pas forcément présentes à un instant donné. Une case de $\mathcal{M}$ ne contient une variable logique que si la variable de flot correspondante a été introduite dans le CSP au cycle considéré ; elle est vide sinon.

Pour coder l'aspect temporel de `Lustre`, deux variables logiques supplémentaires sont introduites. La première, notée C_{max}, exprime le numéro du plus grand numéro de cycle connu dans le CSP courant. En effet, l'opérateur `pre` introduisant une récursion naturelle pour la plupart des définitions de flot, le nombre de cycles nécessaire pour atteindre un objectif est lui-même dépendant du problème. GATeL créant les séquences en arrière, depuis le cycle final vers le cycle initial, C_{max} est initialisé à 0 et incrémenté à chaque fois qu'un nouveau cycle est nécessaire à la résolution. Ainsi les cycles sont numérotés dans l'ordre croissant depuis le cycle final vers le cycle initial. À cause de la progression en arrière, on ne sait pas à l'avance, lorsqu'un cycle est créé, si celui-ci est le cycle initial de la séquence. La seconde variable, notée *Statut*, symbolise cette information pour le plus grand numéro de cycle connu (les autres cycles étant forcément non initiaux). Ce statut intervient dans le traitement de l'opérateur « -> » : cet opérateur est équivalent à son membre gauche quand le statut est initial, et à son membre droit sinon.

L'ensemble V des variables présentes à un instant donné dans le CSP est l'union des variables logiques de $\mathcal{M}$, des deux variables temporelles, et potentiellement de variables supplémentaires permettant de décomposer les expressions complexes vers des contraintes atomiques. Les variables logiques sont introduites à l'aide de la primitive $fresh(D)$, qui retourne une variable libre V telle que $dom(V) = D$.

Domaines. Chaque variable logique de V est attribuée à un domaine $dom(V)$ correspondant à sa déclaration de type. Les variables booléennes sont attribuées dans un intervalle de valeurs symboliques $[true, false]$. Le domaine d'une variable entière est une union d'intervalles croissants $[min_1..max_1] \cup ... \cup [min_n..max_n]$, dont les bornes inférieures et supérieures peuvent être

aussi grandes que nécessaire (en particulier pour coder des entiers sur 32, 64 bits ou autre). Ce domaine permet de capturer les inconsistances proche de zéro dans le cas de multiplications. Si $A = 3 * B$ et $B \neq 0$, alors le domaine de A ne peut pas contenir : -2, -1, 1, 2. Ces trous dans les intervalles peuvent être ensuite décalés en fonction des opérations effectuées. La variable logique *Statut* prend des valeurs dans le domaine symbolique $\{initial, non\_initial\}$, et la variable C_{max} est bornée par un paramètre global pour stopper le déroulement du passé dans le cas de récursions non bornées.

Contraintes. Chaque classe de contraintes est généralement associé à un algorithme de filtrage, ou propagateur, qui se charge de retirer les valeurs des domaines des variables contraintes, tout en garantissant qu'aucune solution n'est écartée du CSP. Nous distinguons deux types de contraintes dans GATeL.

Tout d'abord, les contraintes atomiques, non spécifiques à GATeL, qui décrivent des relations arithmétiques ($+, *, -, \leq$, etc.). Pour chaque contraite numérique $\langle R, \Sigma \rangle$, l'algorithme de filtrage définissant la relation R est une consistance de bornes sur les unions d'intervalles des variables de Σ. Ces contraintes portent uniquement sur une variable résultat et deux variables arguments.

Par ailleurs, nous considérons une contrainte propre à notre méthode de génération, qui met en œuvre l'aspect paresseux de l'approche. Cette contrainte est de la forme $propage(Res, Exp, C)$ où *Res* représente la valuation de l'expression `Lustre` *Exp* au cycle C. L'algorithme de filtrage définissant la relation R de cette contrainte est défini inductivement selon l'opérateur de tête du terme *Exp*. Lorsque ce terme est donné par un opérateur arithmétique, la contrainte délègue sa relation à une contrainte atomique. Lorsqu'il s'agit d'un opérateur booléen ou temporel, si *propage* a suffisemment d'information sur l'expression *Exp*, elle peut introduire de nouvelles contraintes correspondant aux sous-termes de *Exp*. Sinon, la contrainte est retardée jusqu'à ce que le domaine d'une des variables de son ensemble Σ soit réduit. La contrainte *propage* est globale car l'ensemble Σ de variables sur lequel travaille l'algorithme de filtrage n'est pas complètement défini tant que toutes les contraintes atomiques correspondant au terme *Exp* n'ont pas été introduites. Cet ensemble Σ est calculé par une fonction auxiliaire *eval* qui analyse les termes pour retrouver les variables de $\mathcal{M}$, sous la portée de *Exp*, qui sont soit déjà propagées soit des entrées, ou bien la variable *Statut* dans certains cas. La fonction *eval* est décrite dans la section 5.

Considérons la propagation de l'expression « `(E > 0) and S` », où `E` est une entrée et `S` est une sortie, dont le résultat vaut `true`, au cycle 0. Le résultat de l'expression étant vrai, l'opérateur `and` peut introduire deux contraintes correspondant à ses sous-termes :

$$\left\{ \begin{array}{l} \langle propage(true, \texttt{(E > 0)}, 0), \emptyset \rangle, \\ \langle propage(true, \texttt{S}, 0), \emptyset \rangle \end{array} \right\}$$

La propagation de la première contrainte va être déléguée vers la contrainte arithmétique « > » avec la variable E_0 introduite dans $\mathcal{M}$, correspondant à l'entrée `E` au cycle 0. Si de plus `S1` est définie par l'expression « `E' or S'` », le point fixe par propagation de contraintes du CSP est :

$$\left\{ \begin{array}{l} \langle E_0 > 0, \{E_0\} \rangle \\ \langle propage(true, \texttt{(E' or S')}, 0), \{E'_0\} \rangle \end{array} \right\}$$

avec $\mathcal{M}(S, 0) = true$.

Notons qu'aucune branche de la disjonction n'est ajoutée sous forme de contrainte. Supposons que lors de la recherche de solutions, la variable E'_0 soit instanciée à *false*. La contrainte de disjonction est de nouveau filtrée, ce qui raffine le système de contraintes en introduisant la définition de `S'` au cycle 0, puisqu'elle est nécessaire. Ce mécanisme consistant à ne pas introduire systématiquement toutes les contraintes à partir des sous-termes est à la base de l'aspect dynamique de GATeL.

4 Définition inductive

GATeL est implémenté en PLC dans l'environnement `ECLiPSe` [1]. Les règles associées à la sémantique opérationnelle de la contrainte globale *propage* sont présentées dans cette section, sur quelques cas représentatifs. L'opérateur « := » représente l'affectation, par opposition aux tests d'égalité et d'inégalité structurels ($=, \neq$) et à l'unification ($\equiv$). L'opérateur *suspend* permet de placer une contrainte en attente de la réduction du domaine d'une de ses variables.

Variable de flot. Lorsque la valeur d'une variable de flot *id* doit être connue à un cycle donnée, deux cas se présentent (*cf.* règle ci-après). Soit la variable a déjà été propagée, auquel cas sa valeur dans $\mathcal{M}$ est unifiée avec la variable *Res* (ligne 4), soit il faut la propager, et une nouvelle variable logique est créée dans la matrice avec le domaine adéquat, noté Dom_{id} (ligne 5). S'il s'agit d'une sortie, l'expression de définition Exp_{id} de cette variable est propagée avec la variable de résultat *Res* (ligne 7).

```
1  propage(Res, id, cycle)
2  ──────────────────────────────────────
3  si  M(id, cycle) ≠ ∅
4  alors  Res ≡ M(id, cycle)
5  sinon  Res := fresh(Dom_id)
6         M(id, cycle) := Res
7         propage(Res, Exp_id, cycle)
```

Opérateur and. Dans le cas où le résultat est *true*, deux contraintes sont introduites, une pour chacun de ses arguments. Sinon, une évaluation partielle du premier argument est effectuée à l'aide de l'évaluateur d'expressions *eval* (ligne 7). Cette évaluation calcule un éventuel résultat instancié pour cet argument et remonte un ensemble de variables rencontrées au cours de cette évaluation (voir section 5). La principale différence entre *eval* et *propage* est que les autres contraintes du CSP ne peuvent pas se réveiller pendant le calcul du premier. Il s'agit donc d'une évaluation locale. Si le résultat de l'évaluation est *true*, alors deux contraintes sont introduites (lignes 9 et 10). Si l'évaluation est *false*, le résultat de la contrainte de départ est unifié à *false* (ligne 14), provoquant le réveil des contraintes en attente sur une évolution de cette variable. On effectue le cas échéant la même opération pour le second argument. Si aucune réduction n'a pu être effectuée, la contrainte se suspend en attente de l'évolution d'une des variables remontées par l'évaluation partielle, ou du résultat (ligne 29).

```
1  propage(Res, ExpA and ExpB, cycle)
2  ──────────────────────────────────────
3  si  Res = true
4  alors  propage(true, ExpA, cycle)
5         propage(true, ExpB, cycle)
6  sinon
7    (R_A, V_A) := eval(ExpA, cycle)
8    si  R_A = true
9    alors  propage(true, ExpA, cycle)
10          propage(Res, ExpB, cycle)
11   sinon
12     si  R_A = false
13     alors  propage(false, ExpA, cycle)
14            Res ≡ false
15     sinon
16       (R_B, V_B) := eval(ExpB, cycle)
17       si  R_B = true
18       alors  propage(Res, ExpA, cycle)
19              propage(true, ExpB, cycle)
20       sinon
21         si  R_B = false
22         alors  Res ≡ false
23                propage(false, ExpB, cycle)
24         sinon
25           si  Res = false
26           alors  Σ := V_A ∪ V_B
27           sinon
28             Σ := {Res} ∪ V_A ∪ V_B
29             suspend(propage(Res, ExpA and ExpB, cycle), Σ)
```

De la même manière, la contrainte *propage* d'un opérateur conditionnel ne propage pas systématiquement ses arguments. Elle attend de connaitre la valeur de vérité de sa condition auparavant. Néanmoins, l'expression correspondant à la condition est toujours propagée.

Opérateur `pre`. Par hypothèse d'induction, et de par la bonne initialisation des programmes `Lustre` sous test [6], nous supposons que lorsque la contrainte *propage* arrive sur un opérateur `pre` à un cycle c, la propriété $C_{max} > c$ est toujours vérifiée. La contrainte traverse cet opérateur en propageant l'expression retardée au cycle précédent :

```
1  propage(Res, pre(Exp), cycle)
2  ──────────────────────────────────────
3    propage(Res, Exp, cycle + 1)
```

Opérateur d'initialisation. Cet opérateur transcrit l'opérateur « -> » de `Lustre`. Si le plus grand numéro de cycle connu est supérieur au cycle auquel on veut propager cet opérateur, le membre droit est directement propagé (ligne 4) car le cycle courant n'est pas le cycle initial. Sinon, on fait intervenir le statut d'initialité : s'il est connu, alors le sous-terme correspondant est propagé (lignes 7 et 10) ; s'il est variable, une évaluation partielle des sous-termes est effectuée : pour le membre gauche (ligne 12), si le résultat évalué R_I est incompatible avec le résultat attendu *Res*, alors le statut est unifié à *non_initial*, ce qui provoque des réveils pour toutes les contraintes potentiellement en attente sur ce statut ; le numéro du plus grand cycle connu est incrémenté et une nouvelle variable de statut est créée. Enfin, le sous-terme de droite est propagé. Pour le membre droit (ligne 20), si le résultat évalué R_P est incompatible avec *Res*, le statut actuel est unifié à *initial*, provoquant aussi d'éventuels réveils. Le membre gauche est ensuite propagé. Si aucune des évaluations n'est incompatible, alors la contrainte se suspend sur les variables remontées lors de l'évaluation, le résultat et le statut du cycle courant (ligne 28).

```
1  propage(Res, ExpI -> ExpP, cycle)
2  ──────────────────────────────────────
3  si  C_max > cycle
4  alors  propage(Res, ExpP, cycle)
5  sinon
6    si  Statut = initial
7    alors  propage(Res, ExpI, cycle)
8    sinon
9      si  Statut = non_initial
10     alors  propage(Res, ExpP, cycle)
11     sinon
12       (R_I, Σ_I) := eval(ExpI, cycle)
13       si  dom(R_I) ∩ dom(Res) = ∅
14       alors
15         Statut ≡ non_initial
16         Statut := fresh({initial, non_initial})
17         C_max := C_max + 1
18         propage(Res, ExpP, cycle)
19       sinon
20         (R_P, Σ_P) := eval(ExpP, cycle)
```

```
21          si  dom(R_P) ∩ dom(Res) = ∅
22          alors
23            Statut ≡ initial
24            propage(Res, ExpI, cycle)
25          sinon
26            Σ := Σ_I ∪ Σ_P ∪ {Res, Statut}
27            Cstr := propage(Res, ExpI-> ExpP, cycle)
28            suspend(Cstr, Σ)
```

Opérateurs arithmétiques. Nous prenons exemple ici de l'opérateur « + » des entiers. Lorsque la contrainte *propage* rencontre un opérateur arithmétique, elle délègue sa gestion à des contraintes spécifiques capables de réduire les domaines de ses variables selon une consistance de bornes appliquée à l'union des intervalles (ici *plus*, ligne 7). Une gestion paresseuse de ces contraintes ne serait pas efficace car la contrainte *propage* n'aurait que peu d'occasions de réduire des sous-termes (uniquement sur les éléments neutres ou absorbants). De plus, ces contraintes arithmétiques doivent pouvoir travailler au plus vite pour influencer les décisions booléennes ou temporelles. Les contraintes arithmétiques ne travaillant que sur des variables attribuées, les arguments sont systématiquement propagés (lignes 5 et 6). Des variables intermédiaires sont ainsi créées avec le type correspondant au résultat (ici des unions d'intervalles, dont le domaine initial est noté $\mathcal{I}$).

```
1  propage(Res, ExpA + ExpB, cycle)
2  ──────────────────────────────────
3  R_A := fresh(I)
4  R_B := fresh(I)
5  propage(R_A, ExpA, cycle)
6  propage(R_B, ExpB, cycle)
7  plus(Res, R_A, R_B)
```

5 Évaluation partielle

L'évaluation partielle des expressions de flots `Lustre` est une fonction dépendant de la matrice $\mathcal{M}$ et du cycle courant. Elle s'applique aux termes des expressions de flots où les variables `Lustre` ne sont pas forcément connues au moment de l'évaluation. Si l'expression de flot peut-être évaluée à un cycle, elle retourne la valeur du flot selon la sémantique `Lustre`. Lorsque des sous-expressions ne peuvent pas encore être évaluées, elle retourne l'ensemble des variables logiques l'évaluation n'a pas été possible (déjà propagées, ou bien de nouvelles variables associées à des entrées on encore propagées). Cela permet au propagateur d'attendre l'instanciation d'au moins une de ces variables avant de tenter à nouveau l'évaluation.

La règle suivante montre l'évaluation de l'opérateur and. On retrouve la sémantique paresseuse de cet opérateur dans la définition. Si aucune instanciation n'a pu être effectuée, la fonction retourne l'ensemble des variables remontées par l'évaluation des sous-termes.

```
1  eval(Exp_1 and Exp_2, cycle)
2  ──────────────────────────────────
3  (R_1, Σ_1) := eval(Exp_1, cycle)
4  si R_1 = true
5  alors retourner eval(Exp_2, cycle)
6  sinon si R_1 = false
7         alors retourner (false, ∅)
8         sinon (R_2, Σ_2) := eval(Exp_2, cycle)
9               si R_2 = true
10              alors retourner (R_1, Σ_1)
11              sinon si R_2 = false
12                    alors retourner (false, ∅)
13                    sinon Val := fresh({false, true})
14                          retourner (Val, Σ_1 ∪ Σ_2)
```

Les variables logiques issues de l'évaluation de $Exp1$ et de $Exp2$ correspondent à des variables de flots dont la définition a déjà été propagée. Les autres opérateurs sont construits de la même manière. La règle d'évaluation d'une variable de flot *id* à un cycle est définie ainsi :

```
1  eval(id, cycle)
2  ──────────────────────────────────
3  si M(id, cycle) = ∅
4  alors R := fresh(Dom_id)
5         retourner (R, ∅)
6  sinon retourner (M(id, cycle), M(id, cycle))
```

6 Génération de séquences de tests

La génération de séquences de tests consiste à résoudre un CSP initial défini par l'unique contrainte *propage* représentant l'objectif de test. Son résultat vaut *true*, le terme est l'expression booléenne spécifiée par la directive `reach`, et le cycle vaut 0.

Exemple. Reprenons l'exemple de la figure 2 avec l'objectif « `reach end_tempo` ». La seule contrainte du CSP initial est donc propage(true, `end_tempo`, 0). La définition du flot `end_tempo` est « `counter > sel_tempo` ». L'algorithme de filtrage introduit les définitions de `counter` et de `sel_tempo` au cycle 0, en les associant à des variables logiques que l'on note respectivement C_0 et Sel_0. L'inégalité est déléguée à la contrainte $C_0 > Sel_0$.

La variable `counter` est définie par une expression conditionnelle. Ici, la propagation de cette expression ne peut pas savoir quelle branche correspond au résultat. Dans un premier temps, seule la condition est propagée : la variable If_c représente le résultat de la condition « `set and not(start)` ». La propagation de l'opérateur s'arrête ici, car il n'y a pas assez d'information au niveau du terme « and ». Une évaluation partielle de ses sous-termes est faite pour savoir quelles sont les variables logiques déjà propagées dont dépend l'expression. Dans notre cas, seule la variable d'entrée $Start_0$ est remontée, la variable `set` n'étant pas encore propagée au cycle 0.

L'expression and dépend de l'ensemble de variables $\Sigma_0 = \{If_c, Start_0\}$. On note P_0 la contrainte suivante :

$$P_0 = \langle propage(If_c, \texttt{ set and not(start) }, 0), \Sigma_0 \rangle$$

Par ailleurs, l'expression conditionnelle est elle aussi traduite en une contrainte. Pour cela, *eval* est appliquée à chaque branche du `if` pour remonter un ensemble de variables logiques : les ensembles $V_{then} = \{Statut\}$ et $V_{else} = \emptyset$. La contrainte P_1 suivante est suspendue en attente d'une réduction de domaines sur les variables de $\Sigma_1 = \{C_0, If_c, Statut\}$:

$$\begin{aligned} & P_1 = \langle propage(C_0, Exp_1, 0), \Sigma_1 \rangle \\ \textit{avec} \quad & Exp_1 = \texttt{if(}If_c\texttt{,1->pre(counter)+1,0)} \end{aligned}$$

La propagation de la définition de `sel_tempo` est similaire à celle de `counter`, et va provoquer la suspension des deux contraintes P_2 et P_3 suivantes :

$$\begin{aligned} & P_2 = \langle propage(If_s, Exp_2, 0), \Sigma_2 \rangle \\ & P_3 = \langle propage(Sel_0, Exp_3, 0), \Sigma_3 \rangle \\ \textit{avec} \quad & Exp_2 = \texttt{start and not(abort)}, \\ & Exp_3 = \texttt{if(}If_s\texttt{,tempo,0->pre sel\_tempo)} \\ & \Sigma_2 = \{If_s, Start_0, Abort_0\}, \\ \textit{et} \quad & \Sigma_3 = \{If_s, Sel_0, Tempo_0\}. \end{aligned}$$

De plus lors de l'évaluation de ses branches, la fonction *eval* a montré que le domaine de Sel_0 était compris entre 0 et 50, à partir de la restriction donnée à `tempo` dans le modèle. La contrainte d'inégalité représentant l'objectif impose la réduction de la borne inférieure du domaine de C_0 à 1, ce qui provoque le réveil de la contrainte P_1. Lors de la propagation, on constate par évaluation partielle que le domaine de l'expression correspondant à la branche *else* est incompatible avec celui du résultat. Cela impose que la condition If_c soit vraie, ce qui a plusieurs conséquences : Tout d'abord, puisque la condition est vraie, l'expression « `1 -> pre(counter) + 1` » est propagée et unifiée avec le résultat C_0. Une nouvelle contrainte P_4 est ajoutée. Ensuite, la variable If_c étant instanciée avec la valeur *true*, la contrainte P_0 est réveillée et les deux branches de la conjonction sont alors propagées. La première branche va imposer que Set_0 soit vraie puis suspendre sa définition en attente d'informations supplémentaires concernant entre autre la variable $Start_0$; la seconde branche impose justement que $Start_0$ soit faux. Cette dernière information est donc reprise par la contrainte de définition de Set_0. Puisque $Start_0$ est fausse, la fonction *eval* détermine que l'expression `start and not(abort)`, en membre gauche de l'opérateur d'initialisation porté par Set_O, doit être fausse. Comme Set_0 est vraie, il n'est pas possible que le statut du cycle 0 soit initial. La variable *Statut* est donc instanciée à *non_initial*, et C_{max} repoussé au cycle 1. Enfin, ce changement de statut a pour effet de réveiller P_4 et d'introduire la définition de `counter` au cycle 1.

La propagation continue jusqu'à atteindre un point fixe composé de 8 contraintes portant du cycle 0 au cycle 2. GATeL choisit alors une variable et effectue une étape de résolution afin de réveiller certaines contraintes.

Heuristiques de résolution. La résolution permet de faire avancer la génération de séquences de tests lorsqu'un point fixe est atteint, en instanciant une variable logique avec une valeur de son domaine. Le choix de la variable se fait en plusieurs étapes :

1. Récupérer les variables booléennes qui ont le plus de contraintes en attente, et parmi celles-ci, en choisir une au plus grand numéro de cycle.
2. Comparer ce nombre de contraintes avec le nombre de contraintes en attente sur la variable *Statut*.
3. Si ce deuxième nombre est plus grand, chercher à fermer le passé en instanciant le statut à *initial*. Si cette instanciation échoue, le statut est instancié à *non_initial* et la procédure recommencera après le point fixe de propagation qui suivra.
4. Sinon, instancier la variable booléenne de manière aléatoire dans son domaine.
5. Si aucune variable booléenne ni de statut n'est présente dans les contraintes, la procédure choisit une variable entière correspondant à une entrée et l'instancie de manière aléatoire dans son domaine.

La procédure cherche donc en priorité à faire avancer la partie paresseuse de la propagation avant de s'intéresser à la partie arithmétique. En pratique, les problèmes auxquels on se réfère sont très sous-contraints. Cette partie arithmétique de la procédure est assez rarement appelée.

7 Performances

Nous présentons dans cette section quelques résultats sur des études de cas mettant en évidence l'intérêt de la propagation paresseuse. Les comparaisons sont effectuées entre la version courante de GATeL implémentant les définitions précédentes, et une version modifiée dans laquelle les opérateurs booléens propagent systématiquement leurs arguments. Ces expériences ont été réalisées sur un processeur Intel P4 de 3.6GHz avec 1Go de RAM.

Stratégie	Paresseuse	Systématique
Temps d'execution	**2673**	**17941**
Propagation (ms)	2519	8691
Résolution (ms)	154	9250
Nb. de contraintes	**2343**	**2508**

FIG. 4 – Propagation initiale pour `mode_de_fonct`.

Mesures	Mémoire (Ko)	Temps (ms)
1	5443	169
2	5443	170
3	5443	139
4	5444	169
5	5443	160
6	5443	120
Moyenne	5443,17	154,5
Ecart relatif	0,01%	13,33%

FIG. 5 – Résolution de `mode_de_fonct`, cas paresseux.

Mesures	Mémoire (Ko)	Temps (ms)
1	4773	310
2	114356	18330
3	116270	18640
4	4752	300
5	9750	259
6	114780	18170
Moyenne	60826,17	9249,67
Ecart relatif	98,03%	106,12%

FIG. 6 – Résolution de `mode_de_fonct`, cas systématique.

`mode_de_fonct` Cet exemple est extrait d'un système de protection d'un réacteur nucléaire. Il détermine le mode de fonctionnement : normal, dégradé ou arrêt, selon l'observation d'un taux de comptage de neutrons. L'objectif de test est d'observer un passage en arrêt d'urgence. Pour se produire, celui-ci nécessite 1000 cycles de calcul en maintenant certaines entrées dans des intervalles de variations définis.

Le tableau 4 illustre le temps d'exécution de la propagation initiale de l'objectif de test ainsi que le nombre de contraintes résultant pour les deux stratégies. Les tableaux 5 et 6 donnent les temps de résolution nécessaire pour découvrir les 1000 cycles, ainsi que les écarts-types relatifs constatés. Les temps de calcul et la mémoire occupée sont en moyenne nettement meilleurs dans le cas paresseux, et très stables. Notons cependant que certaines exécutions donnent de meilleurs résultats mémoire dans le cas systématique, mais avec une variabilité plus grande. La grande variabilité est liée à l'imperfection des heuristiques en présence d'un grand nombre de contraintes.

`A33_34` Cet exemple est extrait d'un benchmark de systèmes de protection [8]. Il contient des voteurs sur des entrées répliquées respectant des lois de variations temporelles (contraintes de pente) en observant des dépassements de seuils à hystéresis (haut ou bas), pour positionner des conditions de contrôle. L'objectif de test est d'observer deux dépassements de seuils sur les seconds min de 2 groupes de 4 entrées répliquées. Compte-tenu des contraintes de pente, il nécessite la création d'au moins 80 cycles pour atteindre ces seuils.

Le tableau de la figure 7 montre les temps observés pour la propagation initiale de contraintes. Dans le cas de la stratégie systématique, 12443 contraintes ont été propagées en approximativement 40 secondes. La version paresseuse propage quant à elle 8176 contraintes en 4 secondes. Bien que la première dispose de plus de contraintes, aucune des tentatives de résolution n'a permis d'obtenir une solution en un temps raisonnable (abandons au bout de 15 minutes). Le tableau 8 montre les mesures de l'occupation mémoire et du temps d'execution pour l'étape de résolution avec propagations paresseuses. Sur 12 mesures, 2 n'ont pas abouti à un résultat en un temps raisonnable. La moyenne et l'écart-type relatif sont calculés sur les 10 essais réussis.

`CruiseStateMgt` Cet exemple est le codage en flot de données d'un automate de contrôle d'un système de régulation de vitesse. Un objectif de test intéressant est de montrer que certaines sorties de cet automate sont incompatibles. Dans ce cas, GATeL doit montrer qu'il n'existe aucune séquence aboutissant à un tel objectif. Pour cela, nous limitons le nombre de cycles d'exploration à 5. Les résultats présentés dans le tableau 9 donnent le temps nécessaire à l'exploration complète de l'espace de recherche sans obtenir de solution pour chaque stratégie. On observe que la version systématique est nettement plus rapide dans ce parcours. En effet, les flots booléens définissant cet automate sont très fortement corrélés entre eux (partage de sous-expressions communes), ce qui peut être exploité par la stratégie systématique pour réduire l'espace de recherche.

8 Conclusion

Nous avons présenté le mécanisme paresseux d'introduction de contraintes de GATeL. Ce mécanisme s'applique aux opérateurs temporels et booléens pour minimiser le nombre de contraintes et de variables du CSP à un instant donné. Le CSP évolue dynamiquement par restrictions successives dues à l'introduction de nouvelles contraintes au fur et à mesure de la génération de séquences. Nous obtenons des résultats

Mesures	Systématique	Paresseuse
1	40409	3740
2	40409	3710
3	44689	3750
4	40320	3700
5	40299	3700
Moyenne	41225,2	3720
Ecart relatif	4,70%	0,63%

FIG. 7 – Propagation initiale de `A33_34` (ms).

Mesures	Mémoire (Ko)	Temps (ms)
1	37348	108889
2	38782	87809
3	38298	96979
4	31963	40009
5	41016	117609
6	39979	75509
7	38009	85369
8	31309	32399
9	32079	56399
10	43675	103629
Moyenne	37245,8	80460
Ecart relatif	11,21%	36,18%

FIG. 8 – Résolution de `A33_34` dans le cas paresseux.

stables et efficaces pour les exemples industriels dont nous disposons.

Les évolutions récentes du langage Scade visent à introduire la notion d'automate de manière intégrée au paradigme flots de données synchrones. Les automates sont traduits en utilisant de façon intensive la notion d'horloge synchrone sur des types énumérés. Cette extension est en cours d'intégration dans GATeL.

Stratégie	Paresseuse	Systématique
1	333459	77760
2	314890	71400
3	325440	73830
4	309530	74190
Moyenne	320829,75	74295
Ecart relatif	0,03%	0,04%

FIG. 9 – Résolution pour `CruiseStateMgt` (ms).

Références

[1] K. Apt and M. Wallace. *Constraint Logic Programming using ECLiPSe*. Cambridge University Press, 2006.

[2] S. Bardin and P. Herrmann. Structural testing of executables. In *ICST*, pages 22–31. IEEE Computer Society, 2008.

[3] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. The synchronous languages twelve years later. *Proceedings of the IEEE*, 91(1) :64–83, 2003.

[4] F. Bouquet, B. Legeard, and F. Peureux. CLPS-B : A constraint solver to animate a B specification. *International Journal on Software Tools for Technology Transfer, STTT*, 6(2) :143–157, August 2004.

[5] Paul Caspi, Daniel Pilaud, Nicolas Halbwachs, and John A. Plaice. LUSTRE : A declarative language for programming synchronous systems. pages 178–188. ACM Press New York, NY, USA, 1987.

[6] J-L. Colaço and M. Pouzet. Type-based Initialization Analysis of a Synchronous Data-flow Language. *International Journal on Software Tools for Technology Transfer (STTT)*, 6(3) :245–255, August 2004.

[7] Rina Dechter and Avi Dechter. Belief maintenance in dynamic constraint networks. In *7th Conference On Artificial Intelligence*, 1988.

[8] Jean Gassino, Pascal Régnier, Bruno Marre, and Benjamin Blanc. Criteria and Associated Tool for Functional Test Coverage of Safety Critical Software. In *Proceedings of 4th ANS International Topical Meeting on Nuclear Plant Instrumentation, Controls and Human-Machine Interface Technologies (NPIC HMIT 2004)*.

[9] Arnaud Gotlieb, Bernard Botella, and Michel Rueher. Automatic test data generation using constraint solving techniques. *ACM SIGSOFT Software Engineering Notes*, 23(2) :53–62, 1998.

[10] Arnaud Gotlieb, Bernard Botella, and Mathieu Watel. Inka : Ten years after the first ideas. In *19th International Conference on Software and Systems Engineering and their Applications (ICSSEA'06)*, 2006.

[11] Bruno Marre and Benjamin Blanc. Test Selection Strategies for Lustre Descriptions in GATeL. *Electronic Notes in Theoretical Computer Science*, 111 :93–111, 2005.

[12] Aditya P. Mathur. *Foundations of Software Testing*. Pearson Education, 2008.

[13] S. Mittal and B. Falkenhainer. Dynamic constraint satisfaction problems. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, 1990.

[14] Gérard Verfaillie and Narendra Jussien. Constraint solving in uncertain and dynamic environments – a survey. *Constraints*, 10(3) :253–281, 2005.

[15] Gérard Verfaillie and Thomas Schiex. Solution reuse in dynamic constraint satisfaction problems. In *12th Conference On Artificial Intelligence*, 1994.

[16] Nicky Williams, Bruno Marre, and Patricia Mouy. On-the-fly generation of k-path tests for C functions. In *19th International Conference on Automated Software Engineering, 2004.*, pages 290–297, 2004.

Un schéma générique pour intégrer des consistances fortes dans les solveurs de contraintes

Julien Vion **Thierry Petit** **Narendra Jussien**

École des Mines de Nantes, LINA UMR CNRS 6241,
4, rue Alfred Kastler, FR-44307 Nantes, France.
julien.vion@emn.fr, thierry.petit@emn.fr, narendra.jussien@emn.fr

Résumé

Nous présentons un schéma générique pour appliquer les consistances fortes avec les outils standard de programmation par contraintes. Ce schéma est notamment applicable aux solveurs événementiels. Nous proposons d'encapsuler un sous-ensemble des contraintes du modèle dans une contrainte globale. Notre approche permet de spécifier un niveau de consistance différent pour différents sous-ensembles de contraintes d'un même modèle. De plus, nous montrons comment des consistances fortes peuvent être appliquées à différents types de contraintes, dont des contraintes définies par l'utilisateur. Afin de souligner l'efficacité pratique de notre paradigme, nous proposons un nouvel algorithme à gros grain pour établir Max-RPC, nommé Max-RPC^{rm}. Nos expérimentations confirment l'intérêt d'utiliser des consistances fortes au sein des outils de programmation par contraintes, particulièrement lorsqu'on applique un niveau de consistance différent pour différents sous-ensembles de contraintes dans un même réseau.

1 Introduction

Cet article présente un schéma générique pour intégrer des consistances fortes dans les outils existants de résolution de problèmes en programmation par contrainte (PPC), et notamment les solveurs de contraintes événementiels. Nous expérimentons notre approche avec un nouvel algorithme « à gros grain » pour la consistance Max-RPC.

Les techniques les plus courantes pour résoudre des problèmes de satisfaction de contraintes sont basées sur l'application de consistances locales. Les consistances locales suppriment des valeurs qui n'appartiennent à aucune solution. Afin d'appliquer un niveau de consistance donné, des *propagateurs* sont associés aux contraintes. Un propagateur est complet s'il élimine toutes les valeurs qui ne peuvent pas satisfaire la contrainte. Une des raisons du succès de la PPC dans la résolution de problèmes réels est que ces propagateurs sont codés à l'aide d'algorithmes de filtrage qui exploitent la sémantique des contraintes. Ces algorithmes sont le plus souvent issus de techniques efficaces de recherche opérationnelle.

La plupart des solveurs de contraintes sont basés sur un schéma de propagation type AC-5 [20]. On les appelle des solveurs événementiels. Dans un tel schéma, chaque propagateur est appelé lorsque des événements surviennent sur les domaines des variables impliquées dans chaque contrainte. À un nœud donné de l'arbre de recherche, le filtrage est réalisé à l'intérieur de chaque contrainte. Une contrainte reçoit les événements relatifs à ses variables et réalise un filtrage, qui déclenche de nouveaux événements. Le point fixe est obtenu après un cycle de propagation des événements impliquant toutes les contraintes, lorsque plus aucun nouvel événement n'est déclenché. Dans un tel contexte, la consistance d'arc généralisée (GAC) est le niveau de consistance locale le plus élevé. Il correspond au cas où tous les propagateurs sont complets.

Il existe pourtant, dans la littérature, des consistances locales plus fortes que GAC [9, 6]. Elles nécessitent la prise en compte de plusieurs contraintes simultanément pour être appliquées. On considère parfois que ces consistances fortes ne peuvent pas (facilement) être intégrées aux outils de PPC, parmi lesquels, notamment, les solveurs événementiels. Ces outils ne proposent pas ces consistances plus fortes, qui, en conséquence, sont rarement utilisées pour résoudre des problèmes industriels.

Cet article démontre que les consistances fortes sont exclues à tort des outils de PPC. Nous présentons un paradigme générique pour ajouter facilement de telles

consistances aux solveurs de contraintes. Notre idée est de définir une contrainte globale [7, 2, 16], qui encapsule un sous-ensemble du réseau de contraintes initial. Le niveau de consistance souhaité est alors appliqué sur ce sous-ensemble de contraintes. On notera que, généralement, une contrainte globale représente un sous-problème ayant une sémantique bien fixée. Ce n'est pas le cas dans notre approche. La contrainte globale est ici utilisée pour appliquer une technique de propagation particulière sur un sous-ensemble de contraintes. Cette idée est similaire à celle utilisée pour la résolution de problèmes sur-contraints dans [17].

Dans la littérature, Bessière et Régin ont proposé d'augmenter la propagation sur des parties d'un réseau de contraintes, en résolvant des sous-problèmes à la volée [4]. Notre approche permet d'utiliser plusieurs niveaux de consistance locale sur plusieurs sous-ensembles de contraintes dans le même modèle. Notre démarche se rapproche de certains travaux récents sur les CSP binaires, relatifs à des heuristiques ayant pour but d'adapter dynamiquement le niveau de propagation à appliquer lors d'un processus de résolution [18]. Notre approche vise à appliquer efficacement différentes consistances fortes, sans limitation sur l'arité ou la nature des contraintes. Enfin, dans une contrainte globale il est possible de gérer les événements de suppression de la manière la mieux adaptée à la consistance que l'on veut appliquer (gestion orientée par les variables ou par les contraintes). Généralement, les solveurs événementiels ne permettent pas une gestion aussi fine de la propagation.

Nous expérimentons notre méthode avec la consistance forte Max-RPC [8], à l'aide du solveur de contraintes Choco [1]. Nous présentons un nouvel algorithme gros grain pour Max-RPC. Cet algorithme exploite des structures de données « stables au retour arrière », de manière similaire à AC-3^{rm} [15], l'un des algorithmes de consistance d'arc les plus efficaces.

2 Préliminaires

Un *réseau de contraintes* $\mathcal{N}$ est constitué d'un ensemble de variables $\mathscr{X}$, d'un ensemble de domaines $\mathscr{D}$, tels que le domaine $\mathrm{dom}(X) \in \mathscr{D}$ de la variable X soit l'ensemble fini des (au plus d) valeurs que la variable X peut prendre, et d'un ensemble $\mathscr{C}$ de e contraintes qui spécifie les combinaisons de valeurs autorisées pour des ensembles de variables données. Une instanciation I est un ensemble de couples variable/valeur (X, v), noté X_v. I est *valide* ssi pour toute variable X impliquée dans I, $v \in \mathrm{dom}(X)$. Une *relation* R d'arité k correspond à un nombre quelconque d'instanciations de la forme $\{X_a, Y_b, \ldots, Z_c\}$, où $a, b, \ldots, c$ sont des valeurs d'un univers donné. Une *contrainte* C d'arité k est un couple $(\mathrm{scp}(C), \mathrm{rel}(C))$, où $\mathrm{scp}(C)$ est un ensemble de k variables et $\mathrm{rel}(C)$ une relation d'arité k. $I[X]$ est la valeur de X dans I. Pour des contraintes binaires, C_{XY} désigne la contrainte telle que $\mathrm{scp}(C) = \{X, Y\}$. Étant donnée une contrainte C, une instanciation I de $\mathrm{scp}(C)$ (ou d'un sur-ensemble de $\mathrm{scp}(C)$), en considérant alors uniqement les variables de $\mathrm{scp}(C)$), *satisfait* C ssi $I \in \mathrm{rel}(C)$. I est alors *autorisée* par C.

Une *solution* d'un réseau $\mathcal{N}(\mathscr{X}, \mathscr{D}, \mathscr{C})$ est une instanciation I_S de toutes les variables $\mathscr{X}$ telle que (1.) $\forall X \in \mathscr{X}, I_S[X] \in \mathrm{dom}(X)$, et (2.) I_S satisfait toutes les contraintes de $\mathscr{C}$ (I_S est *valide* et *autorisée* par toutes les contraintes de $\mathscr{C}$).

2.1 Consistances locales

Définition 1 (Support, Arc-Consistance)**.** *Soit C une contrainte et $X \in \mathrm{scp}(C)$. Un* **support** *pour la valeur $a \in \mathrm{dom}(X)$ sur C est une instanciation $I \in \mathrm{rel}(C)$ telle que $I[X] = a$. $a \in \mathrm{dom}(X)$ est* **arc-consistante** *par rapport à C ssi elle a un support sur C.*

Pour une contrainte binaire, *i.e.*, impliquant X et Y, le support $I = \{X_a, Y_b\}$ de la valeur X_a peut être caractérisé par la valeur Y_b. On dit que Y_b supporte X_a. Dans ce papier, nous appelons l'arc-consistance AC lorsque toutes les contraintes sont binaires, et GAC dans le cas général.

Définition 2 (Fermeture)**.** *Soit $\mathcal{N}(\mathscr{X}, \mathscr{D}, \mathscr{C})$ un réseau de contraintes, Φ une consistance locale (e.g.,* AC*) et $\mathcal{C}$ un ensemble de contraintes $\subseteq \mathscr{C}$. $\Phi(\mathscr{D}, \mathcal{C})$ est la* fermeture *de $\mathscr{D}$ pour Φ sur $\mathcal{C}$,* i.e.*, l'ensemble des domaines obtenus d'après $\mathscr{D}$ où $\forall X$, toute valeur $a \in \mathrm{dom}(X)$ qui n'est pas Φ-consistante par rapport à une contrainte de $\mathcal{C}$ a été supprimée.*

En ce qui concerne GAC et pour la plupart des consistances, la fermeture est unique. Dans les solveurs de contraintes, un *propagateur* est associé à chaque contrainte afin d'appliquer GAC ou des formes de consistance plus faibles que GAC. Les consistances plus fortes que GAC [9, 6] nécessitent la prise en compte de plus d'une contrainte simultanément pour être appliquées. Cela les a exclues de la plupart des outils de résolution de PPC jusqu'à maintenant.

2.2 Consistance locales fortes

Dans cet article, nous nous intéressons aux consistances de domaine [9], qui suppriment des valeurs des domaines, laissant inchangée la structure du réseau.

Tout d'abord, concernant les réseaux binaires, le concept générique qui capture de nombreuses consistances locales est la (i, j)-consistance [11]. Un réseau de contraintes binaires est (i, j)-consistant ssi il

n'existe pas de domaine vide et si toutes les instanciations consistantes de i variables peuvent être étendues à une instanciation consistante impliquant j variables supplémentaires. Ainsi, l'AC est la $(1,1)$ consistance.

Un réseau de contraintes binaires $\mathcal{N}$ qui n'a pas de domaine vide est *Path Consistent* (PC) ssi il est $(2,1)$-consistant. Il est *Path Inverse Consistent* (PIC) [12] ssi il est $(1,2)$-consistant. Il est *Restricted Path Consistent* (RPC) [3] ssi il est $(1,1)$-consistant et, pour toutes les valeurs a ayant une extension consistante unique b à une variable, (a,b) forme une instanciation $(2,1)$-consistante. $\mathcal{N}$ is *Max-Restricted Path Consistent* (Max-RPC) [8] ssi il est $(1,1)$-consistant et pour chaque valeur X_a et chaque variable $Y \in \mathscr{X} \backslash X$, une extension Y_b de X_a est $(2,1)$-consistante (peut être étendue à une troisème variable). $\mathcal{N}$ est *Singleton Arc-Consistent* (SAC) [9] si chaque valeur est SAC : une valeur X_a est SAC ssi le sous problème obtenu en affectant a à X peut être rendu AC (le principe est similaire au *shaving* qui ne s'applique qu'aux bornes des domaines). $\mathcal{N}$ is *Neighborhood Inverse Consistent* (NIC) [12] ssi n'importe quelle affectation consistante de la valeur a à la variable $X \in \mathscr{X}$ peut être étendue à une instanciation consistante de toutes les variables dans le voisinage de X (voisinage relatif au graphe des contraintes).

Concernant les réseaux non binaires, l'arc-consistance relationnelle et la consistance de chemin relationnelle [10] (relAC et relPC) fournissent les concepts utiles pour étendre les consistances locales des réseaux binaires au cas non binaire. $\mathcal{N}$ est un réseau relAC ssi toute instanciation consistante de toutes les variables sauf une dans une contrainte peut être étendue à cette dernière variable en satisfaisant la contrainte. $\mathcal{N}$ est relPC ssi toute instanciation consistante de toutes les variables sauf une relativement à une paire de contraintes peut être étendue à la dernière variable en satisfaisant ensemble les deux contraintes. À partir de ces notions, de nouvelles consistances pour les réseaux de contraintes généraux inspirées des définitions de PC, PIC et Max-RPC ont été proposées [6]. Enfin, des résultats intéressants ont été obtenus en utilisant la consistance PWC. Un réseau est *Pairwise Consistent* (PWC) [13] ssi aucune relation (instanciations autorisées) n'est vide et si toute instanciation consistante d'une contrainte C peut être étendu de manière consistante à n'importe qu'elle autre contrainte ayant des variables en commun avec C.

Il est possible d'appliquer simultanément PWC et GAC. Cela amène d'autres notions : *Relationally Path Inverse Consistency* (relPIC) and *Pairwise Inverse Consistency* (PWIC) [19].

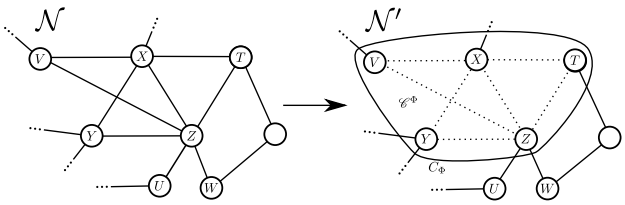


FIG. 1 – Une contrainte globale C_Φ, utilisée pour appliquer une consistance forte sur un sous-ensemble de contraintes $\mathscr{C}^\Phi$. $\mathcal{N}'$ est le réseau obtenu en remplaçant $\mathscr{C}^\Phi$ par la nouvelle contrainte globale.

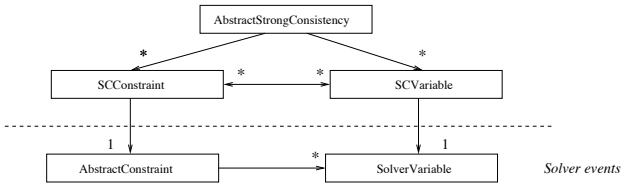


FIG. 2 – Diagramme d'intégration des consistances fortes dans des solveurs événementiels. Les flèches "→" représentent des relations d'agrégation.

3 Une contrainte globale pour les consistances de domaine

Cette section présente un schéma générique (orienté objet) dont le but est d'intégrer les consistances de domaine dans les solveurs de contraintes. Nous montrons aussi sa spécialisation à la consistance Max-RPC. Étant donnée une consistance locale Φ, le principe est de traiter le sous-ensemble de contraintes $\mathscr{C}^\Phi$ sur lequel Φ doit être appliquée. Ce traitement est effectué au sein d'une nouvelle contrainte globale C_Φ, ajoutée au réseau. Les contraintes $\mathscr{C}^\Phi$ sont alors connectées à C_Φ au lieu d'être inclues dans le réseau de contraintes (cf. Figure 1). Les événements relatifs aux contraintes de $\mathscr{C}^\Phi$ sont gérés dans un monde clos, indépendamment de la file de propagation du solveur.

3.1 Un schéma générique

Comme le montre la figure 2, `AbstractStrongConsistency` est la classe abstraite qui sera spécialisée pour implémenter C_Φ, la contrainte globale permettant d'appliquer un niveau de consistance consistance Φ. Le réseau de contraintes correspondant aux contraintes $\mathscr{C}^\Phi$ est stocké à l'intérieur de la contrainte globale. Ainsi, l'intégration de consistances locales au sein d'un solveur événementiel est souple et aisée à mettre en œuvre.

Les contraintes et les variables du réseau d'origine sont encapsulées afin de reconstruire le graphe de contraintes qui correspond aux contraintes $\mathscr{C}^\Phi$,

par l'intermédiaire des classes `SCConstraint` (Strong Consistency Constraint) et `SCVariable` (Strong Consistency Variable). Dans la figure 1, dans $\mathcal{N}'$ toutes les contraintes de $\mathscr{C}^\Phi$ sont déconnectées des variables du solveur. Les variables actives dans la contrainte globale sont encapsulées dans des `SCVariable`s, et les contraintes dans des `SCConstraint`s. Dans $\mathcal{N}'$, la variable Z est connectée aux contraintes C_{UZ}, C_{WZ} et C_Φ du point de vue du solveur. Dans C_Φ, la `SCVariable` Z est connectée via les `SCConstraint`s en pointillés aux `SCVariable`s T, V, X et Y.

3.1.1 Liaison des contraintes

Il est nécessaire d'identifier un dénominateur commun minimal parmi l'ensemble des consistances locales, qui pourra être implémenté en utilisant les services standard fournis par les contraintes du solveur. Dans la figure 2, cela est matérialisé par la classe abstraite `AbstractSolverConstraint`. Dans les solveurs de contraintes (*e.g.*, événementiels), les contraintes sont associées à des propagateurs. Certaines consistances locales, comme SAC, peuvent être directement implémentées à l'aide de propagateurs. Cependant, dans le cas général, les concepts génériques communs à l'ensemble des consistances sont la (i, j)-consistance, relAC et relPC (cf. section 2.2). Aussi, ces consistances sont davantage reliées aux notions d'*instanciation autorisée* et d'*instanciation valide*. Il est nécessaire de pouvoir itérer sur ces instanciations. En outre, les algorithmes qui ont une complexité en temps optimale mémorisent généralement les instanciations qui ont déjà été considérées. Cela implique le fait que les itérateurs délivrent les instanciations toujours dans le même ordre (le plus souvent lexicographique), et qu'il soit possible de démarrer l'itération à partir d'une instanciation quelconque.

Les itérateurs peuvent être implémentés à l'aide de *constraint checkers*. Un constraint checker teste si une instanciation donnée est autorisée par la contrainte, ou si elle ne l'est pas. Cependant, pour de nombreuses contraintes, des itérateurs plus efficaces peuvent être utilisés. Notre schéma propose la possibilité de spécialiser – ou pas – ces itérateurs, via les méthodes `firstSupp` et `nextSupp`. `AbstractSolverConstraint`, une sous-classe de la classe abstraite des contraintes du solveur, spécifie ces méthodes.

Itérateurs génériques. Les services `firstSupp` et `nextSupp` ne sont généralement pas disponibles par défaut dans les solveurs. Nous proposons une implémentation générique basée sur des *constraint checkers*. Cette implémentation est réalisée via un `Adapter` qui spécialise la superclasse `AbstractSolverConstraint`. Cela permet de gérer n'importe quelle contrainte du solveur avec un checker, comme le montre la figure 3.

Algorithm 1: nextSupport(C, I_{fixed}, I) : Instantiation

```
1 I_next ← nextValid(scp(C)\scp(I_fixed), I) ;
2 tant que I_next ≠ ⊥ ∧ ¬check(C, I_next ∪ I_fixed) faire
3 |  I_next ← nextValid(scp(C)\scp(I_fixed), I_next) ;
4 retourner I_next;
```

Algorithm 2: firstSupport(C, I_{fixed}) : Instantiation

```
1 I ← firstValid(scp(C)\scp(I_fixed)) ;
2 si check(C, I ∪ I_fixed) alors retourner I;
3 sinon retourner nextSupp(C, I_fixed, I);
```

Aucune modification n'est donc requise concernant les contraintes du solveur. Notre implémentation des méthodes `firstSupp` et `nextSupp` est donnée par les algorithmes 1 et 2. La complexité en temps est $O(d^{|\text{scp}(C)|-|I_{fixed}|})$. Elle est incrémentale : le processus complet d'itération sur toutes les instanciations autorisées et valides en appelant `firstSupp` et `nextSupp` avec le même paramètre I_{fixed} a la même complexité. Ces fonctions génériques sont principalement adaptées à des contraintes de dureté faible et de basse arité.

Itérateurs spécialisés. Pour certaines contraintes, *e.g.,* les contraintes définies par l'utilisateur, des algorithmes plus efficaces peuvent être utilisés pour `firstSupp` et `nextSupp` (c'est le cas par exemple pour des contraintes arithmétiques, ou encore des contraintes de table positives [5]). Ces algorithmes peuvent ne pas utiliser de constraint checker. La classe `AbstractSolverConstraint` spécialise `SolverConstraint` (cf. Figure 3). Ainsi, il est alors suffisant de spécialiser `AbstractSolverConstraint` pour définir de tels itérateurs.

Utilisation de propagateurs. Certaines consistances fortes comme la consistance de chemin (Path Consistency) peuvent être implémentées directement avec des propagateurs [14]. Notre paradigme permet de réaliser ces implémentations : les propagateurs des contraintes du solveur demeurent disponibles.

3.1.2 Liaison des variables.

Lier les variables est simple, car notre approche ne requiert que des opérations basiques sur les domaines, *i.e.*, itérer sur les valeurs du domaine courant. La classe `SCVariable` permet de représenter le réseau de contraintes $(\text{scp}(C_\Phi), \mathscr{D}, \mathscr{C}^\Phi)$. Un lien est conservé avec les variables du solveur pour effectuer les les modifications des domaines.

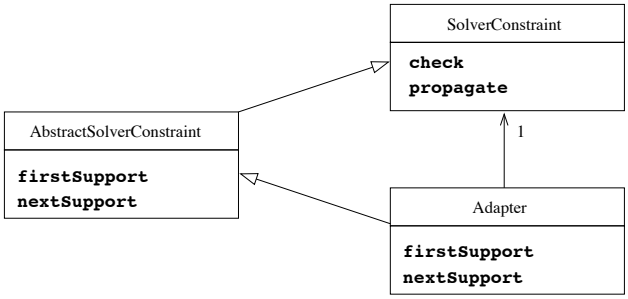


FIG. 3 – Une implementation générique des fonctions d'itération sur des supports, étant données les contraintes fournies par un solveur. Les flèches "—▷" représentent des relations de spécialisation.

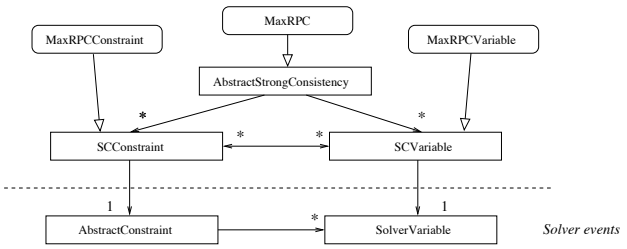


FIG. 4 – Diagramme d'intégration de MaxRPC dans un solveur événementiel.

3.1.3 Heuristiques basées sur le degré des variables.

Certaines heuristiques de choix de variables pour les réseaux binaires, comme $dom/ddeg$ ou $dom/wdeg$, sont liées à la structure du graphe de contraintes. Comme les contraintes $\mathscr{C}^\Phi$ ne sont plus connectées au modèle, elle ne sont plus prises en compte par ces heuristiques. Afin de pallier ce problème dans notre implémentation, nous avons permis à ces heuristiques de connaître directement le score d'une variable (en terme de degré dans le graphe de contraintes) via une requête à la classe `AbstractStrongConsistency`. La contrainte globale est, elle, capable de connaître le degré de chaque variable dans le réseau $\mathscr{C}^\Phi$ (c'est une donnée).

3.2 Une spécialisation concrète : Max-RPC

La figure 4 décrit la spécialisation de notre modèle pour coder la consistance locale forte Max-RPC [8], relative à des réseaux binaires. La classe `MaxRPC` définit la contrainte globale qui sera fournie à l'utilisateur. Elle étend la classe abstraite `AbstractStrongConsistency`, afin d'implémenter l'algorithme de propagation de Max-RPC. Cette consistance locale requiert de connaître les 3-cliques du graphe de contraintes, afin de tester les extensions des instanciations consistantes à n'importe quelle troisième variable. Les classes `SCConstraint` et `SCVariable` sont étendues afin de manipuler efficacement les 3-cliques.

4 Un nouvel algorithme à gros grain pour Max-RPC

Cette section présente Max-RPC^{rm}, un nouvel algorithme à gros grain pour Max-RPC, utilisé en section 5 pour expérimenter notre approche. Cet algorithme exploite des structures de données stables au retour arrière (backtrack-stable) inspirées d'AC-3^{rm} [15]. rm signifie *résidu multi-directionnel*; un résidu est un support qui a été stocké pendant l'exécution de la procédure prouvant qu'une valeur était AC. Lors des appels suivants, cette procédure teste simplement si ce support est toujours valide avant d'en chercher un nouveau. Ces structures de données ne nécessitent pas d'être restaurées ou ré-initialisées lors d'un retour-arrière. En conséquence, le maintien de la structure a un coût faible. Bien qu'étant en théorie non optimal dans le pire des cas, Lecoutre & Hemery ont montré dans [15] que AC-3^{rm} se comporte mieux que les algorithmes d'AC optimaux dans de nombreux cas.

4.1 L'algorithme

« À gros grain » signifie que la propagation dans l'algorithme est gérée soit au niveau d'une variable soit au niveau d'une contrainte, contrairement aux algorithmes comme AC-6 ou GAC-schema qui gèrent la propagation au niveau des valeurs.

4.1.1 Max-RPC^{rm}

Les algorithmes 3 à 6 décrivent Max-RPC^{rm} ; les lignes 6 à 8 de l'algorithme 3 et 5 à 8 de l'Algorithme 5 sont ajoutées à un algorithme AC-3^{rm} standard.

Algorithme 3. Il contient la boucle principale de l'algorithme. Il est basé sur une file contenant les variables ayant été modifiées (*i.e.*, qui ont perdu des valeurs), qui peuvent entraîner la perte des supports dans les domaines des variables voisines. Dans l'exemple de la figure 1 (où l'on ne considère que les contraintes de $\mathscr{C}^\Phi$), si la variable X est modifiée, alors l'algorithme doit tester si toutes les valeurs de T ont encore un support relativement à la contrainte C_{XT}, si toutes les valeurs de V ont un support sur C_{XV}, et de même pour Y et Z. Cela est réalisé via les lignes 4 à 7 de l'algorithme 3. La fonction `revise` décrite par l'algorithme 5 contrôle quant à elle l'existence de tels supports. Elle supprime les valeurs et renvoie **true** ssi il n'y en a pas (et donc **false** si aucune valeur n'a été supprimée).

La variable modifiée qui a été prise en compte peut aussi avoir causé la perte de supports PC pour les

Algorithm 3: MaxRPC($P = (\mathscr{X}, \mathscr{C})$, $\mathscr{Y}$)

$\mathscr{Y}$: the set of variables modified since the last call to MaxRPC

```
1  Q ← Y ;
2  while Q ≠ ∅ do
3      pick X from Q ;
4      foreach Y ∈ X | ∃C_XY ∈ C do
5          foreach v ∈ dom(Y) do
6              if revise(C_XY, Y_v, true) then
7                  Q ← Q ∪ {Y};
8      foreach (Y, Z) ∈ X^2 | ∃(C_XY, C_YZ, C_XZ) ∈ C^3
       do
9          foreach v ∈ dom(Y) do
10             if revisePC(C_YZ, Y_v, X) then
11                 Q ← Q ∪ {Y};
12         foreach v ∈ dom(Z) do
13             if revisePC(C_YZ, Z_v, X) then
14                 Q ← Q ∪ {Z};
```

Algorithm 4: revisePC(C_{YZ}, Y_a, X) : boolean

Y: the variable to revise because PC supports in X may have been lost

```
1  if pcRes[C_YZ, Y_a][X] ∈ dom(X) then
2      return false ;
3  b ← findPCSupport(Y_a, Z_{res[C_YZ, Y_a]}, X) ;
4  if b = ⊥ then
5      return revise(C_YZ, Y_a, false) ;
6  pcRes[C_YZ, Y_a][X] ← b ; return false;
```

contraintes situées du côté opposé de la 3-clique. Dans la figure 1, si X est modifiée, alors les supports de V et Z sur C_{VZ}, les supports de Y et Z sur C_{YZ} et les supports de T et Z sur C_{TZ} doivent être testés. Cela est réalisé par les lignes 8 à 14 de l'algorithme 3 et par la fonction `revisePC` (Algorithme 4).

Algorithme 5. Il itère sur les supports de la valeur X_a à réviser, (ligne 3 et 17), dans la recherche d'une instanciation PC $\{X_b, Y_a\}$. La consistance de chemin de l'instanciation est testée par un appel à `findPCSupport` (Algorithme 6) sur chaque variable Z qui forme une 3-clique avec X et Y. `findPCSupport` retourne soit un support de l'instanciation $\{X_b, Y_a\}$ dans le domaine de Z, soit la valeur $\perp$ si aucun support ne peut être trouvé. Si aucun support PC pour Y_a ne peut être trouvé, la valeur est supprimée et la fonction retourne **true**.

4.1.2 Résidus

La fonction `revise` teste tout d'abord la validité du résidu (ligne 1). Les résidus sont stockés dans la

Algorithm 5: revise(C_{XY}, Y_a, *supportIsPC*) : boolean

Y_a: the value of Y to revise against C_{XY} – supports in X may have been lost

supportIsPC: **false** if one of $pcRes[C_{XY}, Y_a]$ is no longer valid

```
1  if supportIsPC∧ res[C_XY, Y_a] ∈ dom(X) then
2      return false ;
3  b ← firstSupp(C_XY, {Y_a})[X] ;
4  while b ≠ ⊥ do
5      PConsistent ← true ;
6      foreach Z ∈ X | (X, Y, Z) form a 3-clique do
7          c ← findPCSupport(Y_a, X_b, Z) ;
8          if c = ⊥ then
9              PConsistent ← false ;
10             break ;
11         currentPcRes[Z] ← c ;
12     if PConsistent then
13         res[C_XY, Y_a] ← b ; res[C_XY, X_b] ← a ;
14         pcRes[C_XY, Y_a] ← currentPcRes ;
15         pcRes[C_XY, X_b] ← currentPcRes ;
16         return false ;
17     b ← nextSupp(C_XY, {Y_a}, {X_b, Y_a})[X] ;
18 remove a from dom(Y) ;
19 return true ;
```

Algorithm 6: findPCSupport(X_a, Y_b, Z) : value

```
1  c1 ← firstSupp(C_XZ, {X_a})[Z] ;
   c2 ← firstSupp(C_YZ, {Y_b})[Z] ;
2  while c1 ≠ ⊥ ∧ c2 ≠ ⊥ ∧ c1 ≠ c2 do
3      if c1 < c2 then
4          c1 ← nextSupp(C_XZ, {X_a}, {X_a, Z_{c2−1}})[Z] ;
5      else
6          c2 ← nextSupp(C_YZ, {Y_b}, {Y_b, Z_{c1−1}})[Z] ;
7  if c1 = c2 then return c1 ;
8  return ⊥ ;
```

structure de données globale $res[C, X_a]$ (complexité spatiale en $O(ed)$).

L'algorithme utilise aussi des résidus pour les supports PC, stockés dans la structure *pcRes* (complexité en espace $O(cd)$, où c est le nombre de 3-cliques du réseau). L'idée est d'associer le résidu trouvé par la fonction `revise` avec la valeur PC trouvée pour chaque troisième variable dans la 3-clique. De cette façon, à la fin du processus, $(X_a, res[C_{XY}, X_a], pcRes[C_{XY}, X_a][Z])$ forme une 3-clique dans la microstructure du graphe de contraintes pour chaque 3-clique (X, Y, Z) du réseau, et ce pour tout $a \in \text{dom}(X)$.

Dans l'exemple de la figure 5, à la fin du processus, $res[C_{XY}, X_a] = b$, $pcRes[C_{XY}, X_a][Z] = a$,

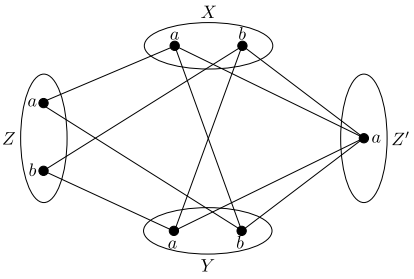


FIG. 5 – Exemple avec deux 3-cliques (microstructure)

$pcRes[C_{XY}, Y_a][Z'] = a$, et ainsi de suite. L'algorithme exploite la bidirectionalité des contraintes : si Y_b est un support pour X_a avec $\{Z_a, Z'_a\}$ comme supports PC, alors X_a est aussi un support pour Y_b avec les mêmes supports PC (cf. lignes 14 et 15 de l'algorithme 5).

Si un support PC obsolète est détecté (ligne 1 de `revisePC`) alors un autre support est recherché. Si aucun ne peut être trouvé, alors le support courant de la valeur courante n'est pas PC, et un autre doit être trouvé (ligne 5 de 4).

4.2 Light-MaxRPC^{rm}

En considérant les consommations élevées en mémoire et en temps causée par la structure de données *pcRes* et les appels successifs de la fonction `revisePC`, nous proposons de modifier l'algorithme 3 en supprimant la boucle **foreach do** des lignes 8 à 14. `revisePC` et *pcRes* n'ont alors plus d'utilité et peuvent être supprimées, ainsi que les lignes 11, 14 et 15 de l'algorithme 5 (il s'agit des parties grisées des algorithmes). L'algorithme ainsi obtenu réalise une approximation de Max-RPC, qui est néanmoins plus forte qu'AC. Il assure que toutes les valeurs qui n'étaient pas Max-RPC avant l'appel à Light-MaxRPC^{rm} sont filtrées.

La consistance qui résulte de l'application de cet algorithme n'est pas monotone. Elle dépend de l'ordre dans lequel les variables sont extraites de $\mathscr{Q}$. Nos expérimentations en section 5 montrent empiriquement que le filtrage de Light-MaxRPC n'est que légèrement plus faible que celui de Max-RPC sur des problèmes aléatoires, alors qu'il permet des gains conséquents en complexité spatiale et temporelle.

4.3 Complexité

Dans la suite de l'article, c désigne le nombre de 3-cliques du graphe de contraintes ($c \leq \binom{n}{3} \in O(n^3)$), g le degré maximal d'une variable, et s le nombre maximal de 3-cliques qui partagent une même (unique) contrainte. Si le graphe de contraintes n'est pas vide (au moins deux variables et une contrainte) alors $s < g < n$. Les complexités sont décrites en termes de tests de contraintes, qui sont supposés constants en temps.

Proposition 1. *Après une phase d'initialisation en $O(eg)$,* MaxRPC^{rm} *a une complexité temporelle dans le pire des cas en $O(ed^3 + csd^4)$.*

Squelette de preuve. La phase d'initialisation détecte et lie les 3-cliques aux contraintes et variables, en $O(eg)$.

La boucle principale de l'algorithme dépend de la file de variables $\mathscr{Q}$. Sachant que les variables sont ajoutées à $\mathscr{Q}$ lorsqu'elles sont modifiées, elles peuvent être ajoutées chacune au plus d fois, impliquant que cette boucle soit exécutée $O(nd)$ fois. Cette propriété reste vraie lorsque l'algorithme est appelé plusieurs fois entre chaque appel, en supprimant une valeur du domaine d'une variable à chaque fois. Cet algorithme est incrémental, notamment lorsqu'on le maintient au sein d'un processus de recherche systématique d'une solution. On considère séparément les deux parties de la boucle principale.

1. La boucle **foreach do** (lignes 4 à 7 de l' algorithme 3).

   Elle peut être exécutée $O(g)$ fois. Dans le pire des cas le réseau entier est exploré de façon homogène, donc la boucle est amortie avec le facteur $O(n)$ de la boucle principale pour une complexité globale en $O(e)$. La boucle **foreach do** des lignes 5 à 7 implique $O(d)$ appels à `revise` (soit au total $O(ed^2)$).

   La fonction `revise` (Algorithme 5) appelle en premier `firstSupp` ($O(d)$ avec l'algorithme 2 sans hypothèse sur la nature des contraintes). La boucle **while do** est réalisée $O(d)$ fois. Les appels à `nextSupp` (Ligne 17) font partie de la boucle. La boucle **foreach do** (lignes 6 à 11) peut être réalisée $O(s)$ fois. Elle implique un appel à `findPC-Support`, en $O(d)$. En conséquence, `revise` est en $O(d + sd^2)$. La complexité globale de cette partie est donc $O(ed^3 + esd^4)$. Le facteur $O(es)$ est amorti en $O(c)$, soit $O(ed^3 + cd^4)$.

2. La boucle **foreach do** (lignes 8 à 14 de l'algorithme 3).

   Le nombre de pas de cette boucle est amorti avec la boucle principale en $O(cd)$. Chaque pas exécute $O(d)$ appels à `revisePC`, dont la complexité dans le pire des cas est plafonnée par `revise` à la ligne 3 de l'algorithme 4. Cette partie de l'algorithme est donc en $O(cd^2.(d + sd^2)) = O(csd^4)$.

La phase d'initialisation et les deux parties entraînent une complexité temporelle en $O(eg + ed^3 + csd^4)$. □

Avec un graphe de contraintes complet, la complexité en temps est $O(n^4d^4)$. Elle peut être comparée à la borne inférieure $O(eg + ed^2 + cd^3)$ ($O(n^3d^3)$ pour un graphe complet) d'un algorithme optimal pour Max-RPC. Si `revise` est appelée à cause de la suppression d'une valeur qui n'apparaît dans aucun support, alors sa complexité est réduite à (sd). En pratique, cela arrive très régulièrement, ce qui explique le bon comportement pratique de notre algorithme. Comme Light-MaxRPC^{rm} supprime la deuxième partie de l'algorithme, sa complexité est $O(eg + ed^3 + cd^4)$. Pour les deux variantes de l'algorithme, la phase d'initialisation en $O(eg)$ est réalisée avant le premier appel à l'algorithme 3, et seulement une fois lorsqu'on maintient Max-RPC pendant la recherche.

Enfin, on peut comparer ces complexités pour maintenir un niveau de complexité strictement supérieur à AC avec celle d'AC-3^{rm} (en $O(n^2d^3)$ pour un graphe complet).

5 Expérimentations

Nous avons implémenté le diagramme de la Figure 4 à l'aide du solveur Choco [1], en utilisant les algorithmes pour Max-RPC décrits précédemment. Dans nos tests, MaxRPC^{rm} et sa variante allégée sont comparés à AC-3^{rm}. Sur les figures, chaque point est le résultat médian relatif à 50 problèmes binaires aléatoires ayant des caractéristiques variables. Un problème aléatoire est caractérisé par un quadruplet (n, d, γ, t), dont les éléments représentent respectivement le nombre de variables, le nombre de valeurs, la densité du graphe[1] et la dureté des contraintes[2].

Pre-processing. La figure 6 compare le temps et la mémoire consommés lors d'une propagation initiale sur des problèmes assez gros (200 variables et 30 valeurs). Dans ces tests, les contraintes formant des 3-cliques sont inclues dans la contrainte globale. Une basse densité du graphe de contraintes entraîne un petit nombre de 3-cliques. Les résultats sont cohérents avec les complexités théoriques. La faible consommation en espace de Light-Max-RPC^{rm} est mise en évidence par les résultats, comparativement à Max-RPC. Si l'on compare ces résultats à ceux de [9], appliquer une consistance forte via une contrainte globale n'entraîne pas de surcoût significatif. L'algorithme AC-3^{rm} du solveur Choco utilise des structures d'allocation paresseuses, ce qui explique la chute de consommation de mémoire après le seuil.

[1] La densité est la proportion de contraintes dans le graphe par rapport au nombre maximum possible de contraintes : $\gamma = e/\binom{n}{2}$.

[2] La dureté est la proportion d'instanciations interdites pour chaque contrainte.

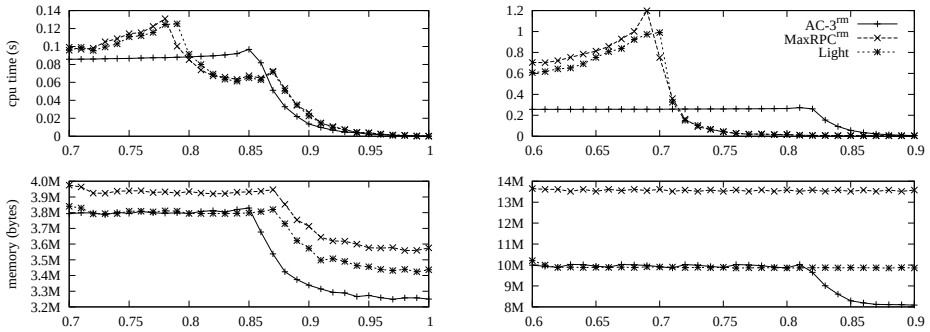
Max-RPC vs AC. La Figure 7 décrit les résultats obtenus avec un algorithme de recherche systématique, où différents niveaux de consistance locale sont maintenus pendant la recherche. L'heuristique de choix de variables est *dom/ddeg* (le processus de pondération des contraintes avec *dom/wdeg* n'est pas défini lorsque plus d'une contrainte peut participer à la suppression de toutes les valeurs d'un domaine). Nous utilisons le problème $(105, 20, 5\%, t)$ comme référence (courbes en haut à gauche) et nous augmentons successivement le nombre de valeurs (en haut à droite), de variables (en bas à gauche) et la densité (en bas à droite). Les résultats dans [8] ont montré que maintenir Max-RPC (à l'aide d'un solveur dédié) est intéressant pour des problèmes de grande taille et de faible densité, comparativement à maintenir l'AC. Les algorithmes d'AC utilisés dans [8] sont maintenant assez obsolètes comparés à AC-3^{rm}. Nos résultats montrent que les nouveaux algorithmes que nous proposons pour Max-RPC sont compétitifs par rapport à AC-3^{rm}, à la fois en termes de nœuds et de temps, sur des problèmes de grande taille et de faible densité.

Combiner des consistances locales. Le tableau 1 montre l'intérêt de la nouvelle possibilité qu'offre naturellement notre approche : combiner deux niveaux de consistance différents pour résoudre un même réseau de contraintes. La première ligne correspond au résultat médian sur 50 instances de problèmes $(35, 17, 44\%, 31\%)$ forcés à être satisfiables. A l'inverse, les *seeds* de $(105, 20, 5\%, 65\%)$ sont choisis de telle sorte que toutes les instances soient insatisfiables. Le premier problème est mieux résolu en utilisant AC-3^{rm}, le deuxième est mieux résolu avec Max-RPC. La troisième ligne décrit les résultat d'un problème qui concatène les deux précédents, en les liant par une unique contrainte additionnelle. Sur les deux dernières colonnes, l'AC est maintenue sur les parties denses du graphe de contraintes, et Max-RPC sur le reste du réseau. L'heuristique de choix de variable *dom/ddeg* entraîne le fait que le problème dense et satisfiable soit résolu en premier, et ensuite que le solveur *thrashe* pour prouver qu'il n'existe pas de solution à cause de la partie du réseau de faible densité.

Combiner les deux niveaux de consistance permet d'améliorer la résolution, ce qui souligne l'intérêt de notre approche. Les deux dernières lignes présentent les résultats obtenus avec de plus gros problèmes, qui confirment les précédents.

6 Conclusion et perspectives

Dans cet article, nous avons présenté un schéma générique pour appliquer des consistances fortes lorsqu'on utilise les outils standard de PPC. Cette ap-



$(200, 30, 5\%, t)$ $(200, 30, 15\%, t)$

FIG. 6 – Propagation initiale : temps cpu et mémoire vs dureté, sur des problèmes aléatoires binaires homogènes (200 variables, 30 valeurs).

		AC	MaxRPC	Light	AC+MaxRPC	AC+Light
$(35, 17, 44\%, 31\%)$	*cpu (s)*	**6.1**	25.6	11.6	non	non
	nodes	21.4k	5.8k	8.6k	applicable	applicable
$(105, 20, 5\%, 65\%)$	*cpu (s)*	20.0	19.4	**16.9**	non	non
	nodes	38.4 k	20.4 k	19.8 k	applicable	applicable
$(35, 17, 44\%, 31\%)$	*cpu (s)*	96.8	167.2	103.2	90.1	**85.1**
$+(105, 20, 5\%, 65\%)$	*nodes*	200.9k	98.7k	107.2k	167.8k	173.4k
$(110, 20, 5\%, 64\%)$	*cpu (s)*	73.0	60.7	**54.7**	non	non
	nodes	126.3k	54.6k	56.6k	applicable	applicable
$(35, 17, 44\%, 31\%)$	*cpu (s)*	408.0	349.0	272.6	284.1	**259.1**
$+(110, 20, 5\%, 64\%)$	*nodes*	773.0k	252.6k	272.6k	308.7k	316.5k

TAB. 1 – Combinaison de deux niveaux de consistance dans un même modèle.

proche permet d'utiliser simultanément plusieurs niveaux distincts de consistance locale pour résoudre un même problème. L'intérêt de cette possibilité est confirmé par nos résultats expérimentaux. De plus, notre paradigme ne pose aucune hypothèse sur la nature et l'arité des contraintes, et permet donc une application des consistances fortes à de nombreux problèmes. Enfin, nous avons proposé Max-RPC^{rm}, un nouvel algorithme « à gros grain » pour Max-RPC.

Nos travaux futurs viseront à appliquer ce paradigme sur des problèmes pratiques concrets, en utilisant d'autres consistances fortes. En outre, étudier les critères pertinents pour décomposer automatiquement un réseau de contraintes afin d'y appliquer différents niveaux de consistance constitue une perspective très intéressante.

Remerciements. Ces travaux ont été réalisés dans le cadre du projet CANAR (ANR-06-BLAN-0383-02). Les auteurs tiennent à remercier les relecteurs anonymes pour leur aide.

Références

[1] Choco : An open source Java CP library. `http://choco.emn.fr/`, 2008.

[2] N. Beldiceanu, M. Carlsson, and J.-X. Rampon. Global constraint catalog. Technical Report T2005-08, SICS, 2005.

[3] P. Berlandier. Improving domain filtering using restricted path consistency. In *Proc. IEEE-CAIA'95*, 1995.

[4] C. Bessière and J.-C. Régin. Enforcing arc consistency on global constraints by solving subproblems on the fly. In *CP*, pages 103–117, 1999.

[5] C. Bessière and J.-C. Régin. Arc consistency for general constraint networks : preliminary results. In *Proc. IJCAI'97*, pages 398–404, 1997.

[6] C. Bessière, K. Stergiou, and T. Walsh. Domain filtering consistencies for non-binary constraints. *Artificial Intelligence*, 172(6-7) :800–822, 2008.

[7] C. Bessière and P. van Hentenryck. To be or not to be... a global constraint. In *Proc. CP'03*, pages 789–794. Springer, 2003.

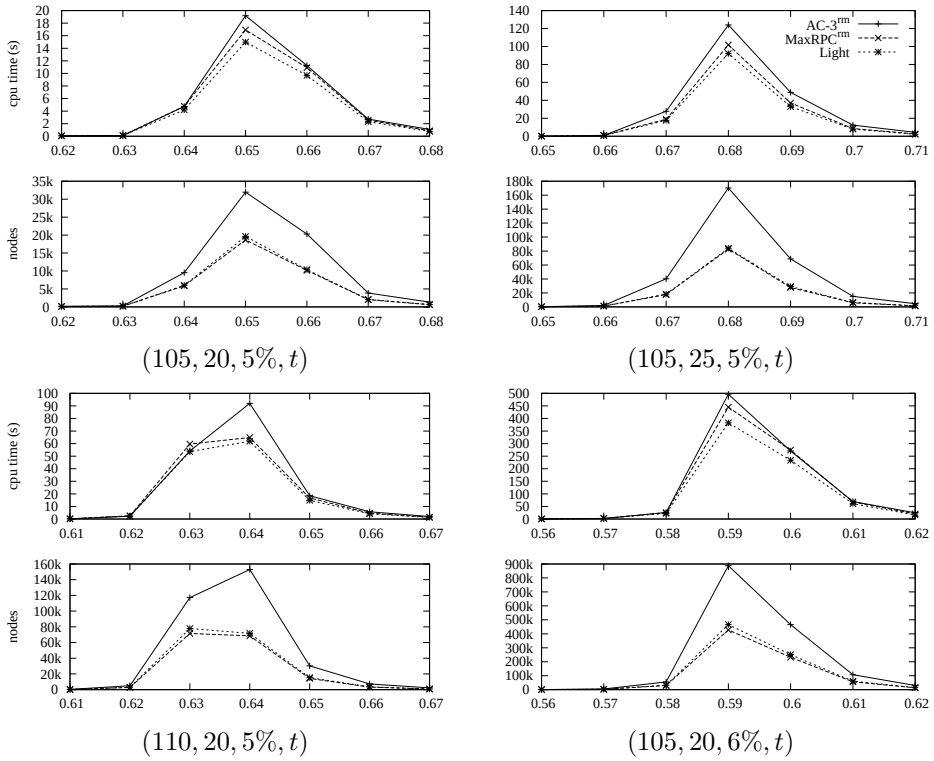


FIG. 7 – Recherche d'une solution : temps cpu et nœuds vs dureté, sur des problèmes aléatoires binaires homogènes (105-110 variables, 20-25 valeurs).

[8] R. Debruyne and C. Bessière. From restricted path consistency to max-restricted path consistency. In *Proc. CP'97*, pages 312–326, 1997.

[9] R. Debruyne and C. Bessière. Domain filtering consistencies. *Journal of Artificial Intelligence Research*, 14 :205–230, 2001.

[10] R. Dechter and P. van Beek. Local and global relational consistency. *Theoretical Computer Science*, 173(1) :283–308, 1997.

[11] E.C. Freuder. A sufficient condition for backtrack-free search. *Journal of the ACM*, 29(1) :24–32, 1982.

[12] E.C. Freuder and C.D. Elfe. Neighborhood inverse consistency preprocessing. In *Proc. AAAI'96*, pages 202–208, 1996.

[13] P. Janssen, P. Jégou, B. Nouguier, and M.C. Vilarem. A filtering process for general constraint-satisfaction problems : achieving pairwise-consistency using an associated binary representation. In *Proc. of IEEE International Workshop on Tools for Artificial Intelligence*, pages 420–427, 1989.

[14] C. Lecoutre, S. Cardon, and J. Vion. Path Consistency by Dual Consistency. In *Proc. CP'07*, pages 438–452, 2007.

[15] C. Lecoutre and F. Hemery. A study of residual supports in arc consistency. In *Proc. IJCAI'07*, pages 125–130, 2007.

[16] J.-C. Régin. A filtering algorithm for constraints of difference in CSPs. In *Proc. AAAI'94*, pages 362–367, 1994.

[17] J-C. Régin, T. Petit, C. Bessière, and J.-F. Puget. An original constraint based approach for solving over constrained prolems. In *Proc. CP'00*, pages 543–548, 2000.

[18] K. Stergiou. Heuristics for dynamically adapting propagation. In *ECAI*, pages 485–489, 2008.

[19] K. Stergiou and T. Walsh. Inverse consistencies for non-binary constraints. In *Proc. ECAI'06*, volume 6, pages 153–157, 2006.

[20] P. van Hentenryck, Y. Deville, and C.M. Teng. A generic arc-consistency algorithm and its specializations. *Artificial Intelligence*, 57 :291–321, 1992.

Consistance duale et réseaux non-binaires

Julien Vion

École des Mines de Nantes, LINA UMR CNRS 6241,
4, rue Alfred Kastler, FR-44307 Nantes, France.
julien.vion@emn.fr

Résumé

La Consistance Duale (DC) a été introduite par Lecoutre, Cardon et Vion dans [10, 11]. Il s'agit d'une nouvelle manière de gérer la Consistance de Chemin (PC), à la définition plus simple et permettant d'introduire de nouvelles approximations et algorithmes particulièrement efficaces en pratique. Un des points intéressants de cette définition est la possibilité de généraliser PC aux réseaux de contraintes (CNs) non-binaires, tout en conservant la nature des contraintes initiales du problème, et en exploitant leurs propagateurs. Cette généralisation permet notamment de voir la Consistance Duale/de Chemin comme un moyen simple et efficace de générer des contraintes binaires implicites directement à partir des propagateurs des contraintes du problème. Cet article montre les implications en termes de complexité de cette généralisation. Des résultats expérimentaux préliminaires montrent le potentiel pratique d'une génération à la volée de telles contraintes implicites, et met en évidence les faiblesses potentielles de la méthode. Des idées prospectives amenées à gérer ces faiblesses sont alors proposées.

Les consistances sont des propriétés des réseaux de contraintes (*Constraint Networks*, CNs), utilisées pour identifier et éliminer, généralement en temps et espace polynomiaux, des valeurs ou des instanciations globalement inconsistantes, c'est-à-dire ne pouvant appartenir aux solutions du CSP associé au CN. Elles sont essentielles dans le processus de résolution d'un CSP et une des principales raisons du succès de la programmation par contraintes [8, 2].

La consistance la plus utile est la consistance d'arc généralisée (GAC), qui permet de détecter des valeurs inconsistantes pour une contrainte donnée. La consistance de chemin (PC) est une des consistances les plus anciennes. Elle est définie sur les CNs binaires, et généralement appliquée aux graphes de contraintes complets [14]. La consistance duale (DC) est une définition alternative (équivalente), plus simple de PC. Elle a été utilisée dans [10] pour définir une nouvelle approximation de PC, nommée DC *conservative* (CDC), et dans [11] pour introduire de nouveaux algorithmes efficaces pour établir PC sur des graphes complets de contraintes binaires.

Dans cet article, nous proposons d'étendre la définition de (C)DC aux CNs non-binaires. DC peut être alors vue comme une technique simple pour déduire automatiquement des contraintes binaires implicites à partir des domaines et contraintes initiaux du CN. Les contraintes implicites sont des contraintes qui peuvent être ajoutées au CN sans en changer l'ensemble des solutions, mais permettant d'améliorer la propagation des décisions. Après les définitions et notations, nous rappelons les caractéristiques des algorithmes de GAC à gros grain, et présentons la consistance duale. Nous décrivons comment appliquer DC à des réseaux non-binaires de contraintes hétérogènes utilisant des propagateurs basés sur la sémantique des contraintes. Nous proposons l'algorithme sDC^{clone}, permettant d'établir DC sur des réseaux de contraintes quelconques en générant des contraintes implicites « à la volée », et qui exploite totalement l'incrémentalité des algorithmes de GAC sous-jacents pour obtenir une complexité temporelle dans le pire des cas en $O(ne_id^3 + nd\psi)$ ($O(n(e+e_i)d^3)$ pour des CNs binaires). Cet algorithme a une meilleure complexité que les algorithmes proposés précédemment[1]. sDC-2.1, une variante généralisée et optimisée de sDC-2 est ensuite proposée, et sa complexité étudiée.

Finalement, nous identifions expérimentalement les défauts de DC, pouvant empêcher son utilisation sur des problèmes industriels. En perspective de ces travaux, nous proposons des approximations de DC permettant de gérer ses faiblesses identifiées.

[1] ψ est la complexité amortie pour établir GAC sur le CN de manière incrémentale, et e_i est le nombre de contraintes implicites générées.

1 Préliminaires

1.1 CN et CSP

Un *réseau de contraintes* (CN) P consiste en un ensemble de n variables $\mathscr{X}$ et un ensemble de e contraintes $\mathscr{C}$. Un domaine, associé à chaque variable X et noté $\text{dom}^P(X)$, est un ensemble fini d'au plus d valeurs que la variable peut prendre dans le CN P. Quand ce sera possible sans ambigüité, nous noterons simplement $\text{dom}(X)$ au lieu de $\text{dom}^P(X)$. Les contraintes spécifient les combinaisons autorisées de valeurs pour des ensembles de variables donnés. Une instanciation I est un ensemble de couples variable/valeur, (X, v), notés X_v, avec v une valeur d'un univers donné U ($\forall X \in \mathscr{X}, \text{dom}(X) \subseteq U$). I est *valide* par rapport à un CN P ssi pour toute variable X impliquée dans I, $v \in \text{dom}^P(X)$.

Une *relation* R est un ensemble d'instanciations. Une *contrainte* C d'arité r est un couple $(\text{scp}(C), \text{rel}(C))$, avec $\text{scp}(C)$ un ensemble de r variables et $\text{rel}(C)$ une relation d'arité r. k est l'arité maximale des contraintes dans un CN donné. Pour une contrainte donnée C, une instanciation I de $\text{scp}(C)$ (ou d'un sur-ensemble de $\text{scp}(C)$, ne considérant dans ce cas que les variables de $\text{scp}(C)$) *satisfait* C ssi $I \in \text{rel}(C)$. On dit que I est *autorisée* par C. Pour contrôler si une instanciation satisfait une contrainte, on effectue un *test de contrainte*. Une instanciation I est *localement consistante* ssi elle est valide et autorisée par toutes les contraintes du CN. Une *solution* d'un CN $P(\mathscr{X}, \mathscr{C})$ est une instanciation localement consistante de toutes les variables de $\mathscr{X}$.

Un CSP est le problème de décision consistant à déterminer si une solution à un CN donné existe. Une instanciation est *globalement consistante* ssi elle est un sous-ensemble d'au moins une solution. Les instanciations qui ne sont pas globalement consistantes sont également appelées *no-goods*. Déterminer si une instanciation localement consistante est un no-good est NP-complet dans le cas général, mais les propriétés de consistance sont utilisées pour identifier des no-goods en utilisant des algorithmes polynomiaux. Étant donnée une consistance Φ, elle peut être *établie* sur P en utilisant un « algorithme de Φ-consistance », dont l'objectif est de détecter et supprimer toutes les instanciations Φ-inconsistantes jusqu'à l'obtention d'un point fixe (une *fermeture* de P par Φ est obtenue). Le plus souvent, le point fixe est unique. Le CN obtenu à partir de P en établissant la consistance Φ sera noté $\Phi(P)$.

1.2 Consistance d'arc généralisée

La consistance d'arc généralisée (GAC) est la propriété de consistance la plus commune et la plus utile.

Algorithm 1: GAC($P = (\mathscr{X}, \mathscr{C})$, $\mathscr{A}$)

P: le CN à filtrer
$\mathscr{A}$: un ensemble initial d'arcs à réviser

```
1 Q ← A
2 tant que Q ≠ ∅ faire
3     prendre (C, X) de Q
4     si revise(C, X) alors
5         si dom(X) = ∅ alors retourner ⊥
6         Q ← Q ∪ mod({X}, C)\(C, X)
7 retourner P
```

Il s'agit d'une propriété de *consistance de domaine* [2], c'est-à-dire qu'elle identifie des no-goods de taille 1 (des valeurs globalement inconsistantes). Les propriétés de consistance qui identifient des no-goods de plus grande taille sont appelées consistances de *relation*, puisqu'elles permettent de supprimer des instanciations des relations des contraintes.

Definition 1 (Consistance d'arc généralisée)**.** Étant donné un CN $P = (\mathscr{X}, \mathscr{C})$:

1. $X_a \mid X \in \mathscr{X}$ et $a \in \text{dom}(X)$ est GAC par rapport à la contrainte $C \in \mathscr{C}$ ssi $\exists I \mid X_a \in I \wedge I$ est valide et autorisée par C. I est alors appelée un *support* de a pour C.
2. X_a est GAC ssi $\forall C \in \mathscr{C} \mid X \in \text{scp}(C)$, X_a est GAC par rapport à C.
3. P est GAC ssi $\forall X \in \mathscr{X}$, $\forall a \in \text{dom}(X)$, X_a est GAC.

Notation 1. L'ensemble $\{(C, Y) \mid \{X, Y\} \subseteq \text{scp}(C) \wedge X \in \mathscr{X} \wedge C \in \mathscr{C}\}$ des arcs à réviser après une modification effectuée sur les variables de l'ensemble $\mathscr{X}$, ou sur les contraintes de l'ensemble $\mathscr{C}$, sera noté $\text{mod}(\mathscr{X}, \mathscr{C})$.

L'algorithme 1 présente la boucle principale des algorithmes de GAC à gros grain, i.e. des variantes de GAC-3. Les variantes résident dans la nature de la fonction `revise`, appelée à la ligne 4 de l'algorithme. Cette version de l'algorithme est « orientée arcs » : la file de propagation contient tous les arcs qui doivent être révisés. Un arc est un couple (C, X), avec $X \in \text{scp}(C)$. Dans un CN donné, on peut définir jusqu'à $O(ek)$ arcs. Un arc (C, X) doit être révisé s'il existe une possibilité pour X de ne pas être AC par rapport à C. Si l'on n'a aucune information sur l'état du CN, tous les arcs doivent être placés dans la file et être ainsi révisés au moins une fois. Si l'on sait qu'une unique variable X a été modifiée dans un CN GAC, seuls les arcs $\text{mod}(\{X\}, \mathscr{C})$ doivent être insérés. L'algorithme 1 peut être initialisé à partir d'un ensemble d'arcs $\mathscr{A}$ (ligne 1). En utilisant la notation 1, GAC

peut être établie sur un CN donné $P = (\mathscr{X}, \mathscr{C})$ par un appel $P \leftarrow \text{GAC}(P, \text{mod}(\mathscr{X}, \mathscr{C}))$.

La complexité dans le pire des cas de ces algorithmes vient du fait qu'un arc donné (C, X) n'est inséré dans la file de propagation que lorsque l'une des $O(k)$ variables de scp(C) est modifiée (il faut alors contrôler si les supports de X par rapport à C sont toujours valides). Or, une variable ne peut être modifiée que d fois. Ainsi, $O(ek^2d)$ appels à `revise` peuvent être effectués dans le pire des cas. L'algorithme de base pour `revise` effectue une itération sur le produit Cartésien des domaines des variables de scp(C) jusqu'à trouver un support pour chaque valeur des domaines, d'où une complexité en $O(kd^k)$ pour `revise` (on suppose qu'un test de contrainte se fait en $O(k)$), et de $O(ek^3d^{k+1})$ pour établir GAC. L'idée de GAC-2001 [5] est de rendre la fonction `revise` incrémentale, de sorte que la complexité amortie de tous les appels à `revise` pour une contrainte donnée est en $O(k^2d^k)$,[2] d'où une complexité en $O(ek^2d^k)$. GAC-schema [4] atteint la complexité optimale $O(ekd^k)$ en utilisant des files de propagation à *grain fin* et en exploitant la multi-directionnalité des contraintes. Cependant, cela nécessite des structures de données additionnelles, et l'algorithme n'est pas toujours plus performant en pratique. D'autres travaux ont indiqué que d'autres variantes sous-optimales de GAC-3, comme GAC-3^{rm} [12] ou GAC-watched [9] sont les plus efficaces en pratique quand ils sont maintenus au cours de la recherche, grâce à l'exploitation de structures de données *stables au backtrack*. Dans le cas binaire ($k = 2$), (G)AC-2001 est optimal en $O(ed^2)$. Finalement, dans le cas binaire, il existe des algorithmes hautement optimisés comme AC-3^{bit} [13], permettant la propagation efficace d'un grand nombre de contraintes binaires en extension.

Toutes les variantes des algorithmes de GAC sont incrémentales : une fois le point fixe atteint dans un CN donné, la complexité amortie dans le pire des cas pour plusieurs appels à l'algorithme, en supprimant au moins une valeur entre chaque appel, est la même que la complexité d'un seul appel. Pour appliquer GAC-3 incrémentalement, il faut veiller à ne réviser un arc que si et seulement si une valeur a été supprimée dans une variable impliquée par la contrainte de l'arc.

1.3 Propagateurs

L'idée des *propagateurs* provient du schéma de propagation générique AC-5 [19]. Les algorithmes de GAC génériques présentés dans la section précédente sont tous exponentiels en l'arité des contraintes, et ne sont pas applicables pour des contraintes de grande arité. Cependant, en pratique, on peut souvent établir GAC efficacement pour une contrainte non-binaire en exploitant les propriétés sémantiques de celle-ci [1, 16]. AC-5 permet ceci en *abstrayant* la fonction `revise`, qui peut alors être spécialisée pour chaque type de contrainte. Ces fonctions `revise` spécialisées sont souvent appelées *propagateurs*. Dans ce contexte, les algorithmes de GAC généraux sont le plus souvent utilisés pour propager des contraintes définies en *extension*, c'est-à-dire à partir d'une liste exhaustive d'instanciations interdites[3], ou pour des contraintes définies en *intention* et utilisant des opérateurs scalaires.

La plupart des travaux sur la résolution générique des CSP se sont focalisés sur l'utilisation de contraintes homogènes, le plus souvent binaires en extension. En pratique, les problèmes industriels mettent en jeu des contraintes hétérogènes, associées à des propagateurs efficaces basés sur leur sémantique. Dans cet article, nous ne faisons aucune hypothèse sur la manière dont GAC doit être établie. Toutes les contraintes pour lesquelles il existe un propagateur spécifique sont conservées, et la sémantique de celles-ci est exploitée.

Par la suite, nous noterons $O(\phi)$ la complexité dans le pire des cas pour établir GAC sur un réseau de contrainte donné, $O(\psi)$ la complexité amortie pour établir GAC *incrémentalement* sur ce réseau, en supprimant au moins une valeur entre chaque propagation, et $O(\rho)$ la complexité spatiale totale du réseau et de tous les propagateurs impliqués. Dans un CN général utilisant GAC-schema comme algorithme de propagation, $O(\phi) = O(\psi) = O(ekd^k)$ et $O(\rho) = O(nd + ekd)$.

1.4 Consistance duale

La consistance duale (DC) introduite dans [10], est une définition alternative (équivalente) à celle de la consistance de chemin [14] :

Definition 2 (Consistance duale)**.** Étant donné un CN $P = (\mathscr{X}, \mathscr{C})$, une instanciation binaire $I = \{X_a, Y_b\}$ est DC ssi $X_a \in \text{AC}(P|_{Y=b}) \wedge Y_b \in \text{AC}(P|_{X=a})$.

P est DC ssi toutes les instanciations binaires localement consistantes de P sont DC.

P est fortement DC (sDC) ssi il est à la fois AC et DC.

L'algorithme connu le plus efficace pour établir la consistance duale forte est sDC-2, décrit dans [11]. Lecoutre *et al.* montrent qu'établir sDC avec cet algorithme a une complexité temporelle dans le pire des cas en $O(n^5d^5)$, nettement supérieure aux meilleures

[2] Le facteur additionnel k provient du fait que la multi-directionnalité des relations ne peut être exploitée, et chaque instanciation peut être prise en compte jusqu'à k fois.

[3] Dans le cas d'une liste d'instanciations *autorisées*, on peut utiliser d'autres algorithmes comme STR [17].

complexités connues pour PC, soit $O(n^3d^3)$. Cependant, le pire des cas pour sDC-2 n'a que très peu de chances d'apparaitre, et l'algorithme est en pratique plus rapide que les algorithmes de PC état-de-l'art sur la plupart des instances de CSP. De plus, sDC-2 n'utilise qu'une structure de données très légère (en $O(n)$).

2 Consistance duale et réseaux non-binaires : algorithmes et complexités

Les algorithmes existants pour PC nécessitent que le CN soit un graphe complet de contraintes binaires, supportant la composition. La définition de DC permet d'établir les remarques suivantes :

1. En remplaçant simplement AC par GAC dans la définition 2, DC peut être appliquée à un CN d'arité quelconque.
2. Les algorithmes de DC ne font aucune hypothèse sur la manière dont (G)AC doit être établi. Les algorithmes peuvent donc utiliser tous les propagateurs état-de-l'art (basés sur la sémantique ou pour les contraintes en extension) fournis par le solveur.
3. Pour établir sDC, il « suffit » d'être en mesure de stocker et exploiter des no-goods binaires, et ce même si le CN original inclut des contraintes non-binaires. Ceci ne suppose pas nécessairement de générer un graphe complet de contraintes (seulement dans le pire des cas).[4]

La dernière proposition provient d'un nouveau point de vue sur DC (et PC) : les contraintes additionnelles introduites pour établir DC sont des contraintes qui sont *impliquées* par les contraintes initiales du problème. DC nous permet de déduire ces contraintes de manière purement sémantique, en exécutant les propagateurs des contraintes d'origine.

Trois algorithmes différents, sDC-1, sDC-2 et sDC-3, établissant la consistance duale forte sur des graphes complets de contraintes binaires, sont décrits dans [11]. Nous proposons d'étendre sDC-1 et sDC-2 pour prendre en compte les réseaux de contraintes non-binaires (sDC-3 est spécifiquement destiné aux réseaux binaires et les résultats expérimentaux montrent qu'il s'agit de l'algorithme le moins intéressant en pratique). Nous exploitons l'incrémentalité des algorithmes de propagation pour obtenir une meilleure borne théorique de complexité temporelle. Nous nous inspirons pour cela de l'algorithme SAC-OPT [3]. Les algorithmes de sDC (et SAC) fonctionnent à partir des *tests de singleton* : pour chaque valeur X_v d'un CN

[4] C'est également vrai pour les algorithmes de PC, mais cela n'a à notre connaissance pas été expérimenté dans des travaux antérieurs.

Algorithm 2: sDC-1($P = (\mathscr{X}, \mathscr{C})$)

```
1  P ← GAC(P, mod(𝒳, 𝒞))
2  marque ← X ← first(𝒳)
3  tant que X = marque faire
4      si |dom(X)| > 1 ∧ checkVar-1(P, X) alors
5          P ← GAC(P, mod({X}, 𝒞))
6          si P = ⊥ alors retourner ⊥
7          marque ← X
8      X ← next(𝒳, X)
9  retourner P
```

Algorithm 3: checkVar-1($P = (\mathscr{X}, \mathscr{C})$, X)

```
1  modif ← faux
2  pour ch. a ∈ dom(X) faire
3      P' ← GAC(P|_{X=a}, mod({X}, 𝒞))
4      si P' = ⊥ alors
5          retirer a de dom^P(X)
6          modif ← vrai
7      sinon
8          pour ch. Y ∈ 𝒳\X faire
9              soit C t.q. C ∈ 𝒞_i ∧ scp(C) = {X, Y}
10             pour ch. b ∈ dom^P(Y) | b ∉
               dom^{P'}(Y) ∧ {X_a, Y_b} ∈ rel(C) faire
11                 retirer {X_a, Y_b} de rel(C)
12                 modif ← vrai
13 retourner modif
```

P, v est affectée à X ($\mathrm{dom}(X)$ est réduit au *singleton* $\{v\}$), et (G)AC est établi sur le CN résultant, noté $P|_{X=v}$. SAC supprime la valeur du domaine de la variable ssi une inconsistance est détectée. DC va plus loin en enregistrant des informations pouvant être déduites du test de singleton sous forme de no-goods binaires qui sont ajoutés au CN en modifiant et/ou en insérant des contraintes implicites. Le sous-ensemble des contraintes implicites sera noté $\mathscr{C}_i$ ($\mathscr{C}_i \subseteq \mathscr{C}$). Toutes les contraintes binaires en extension (ou pouvant efficacement être converties en contraintes en extension) du réseau initial peuvent immédiatement être placées dans $\mathscr{C}_i$. On notera $e_i = |\mathscr{C}_i|$. On a $e_i \leq \binom{n}{2} \in O(n^2)$. D'après la preuve dans [3] sur la globalité des supports pour SAC, toute modification effectuée lors d'un test de singleton doit entrainer le contrôle de tous les singletons impliquant les autres variables.

2.1 sDC-1 : l'approche naïve.

sDC-1, déjà décrit dans [11], est l'algorithme le plus « naïf » pour établir sDC. Les algorithmes 2 et 3 dé-

crivent sDC-1. L'algorithme itère sur toutes les valeurs des domaines des variables. La *marque* et les fonctions `first`/`next` utilisées dans l'algorithme 2 sont utilisées pour parcourir les variables du CN en boucle, jusqu'à ce que toutes les variables aient été testées par `checkVar-1` sans qu'aucune modification n'ait été effectuée. `next`$(\mathscr{X}, X)$ considère $\mathscr{X}$ comme un ensemble ordonné, et renvoie soit la variable juste après X dans $\mathscr{X}$, soit la première variable de $\mathscr{X}$ (`first`$(\mathscr{X})$), ssi X était la dernière variable de $\mathscr{X}$. Les tests de singleton sont effectués dans la boucle principale **pour ch. faire** de `checkVar-1` (algorithme 3). La seconde boucle, lignes 8-12 permet de stocker les no-goods pouvant être déduits à partir de l'établissement de GAC à la ligne 3. À la ligne 9, la contrainte est créée « à la volée » ssi elle n'existe pas. Les contraintes créées sont initialement des contraintes universelles, qui autorisent toutes les instanciations des variables.

Proposition 1. Appliqué à un CN quelconque, sDC-1 a une complexité temporelle dans le pire des cas en $O(ne_i^2d^5 + ne_id^3\phi)$ et une complexité spatiale en $O(e_id^2)$.

Ébauche de preuve. $O(nd)$ singletons doivent être testés, et si n'importe quel test de singleton entraine une modification, tous les singletons doivent à nouveau être testés [3]. La plus petite modification pouvant survenir est la suppression d'un no-good binaire dans la relation d'une contrainte implicite. Il peut ainsi y avoir jusqu'à $O(e_id^2)$ modifications. Un test de singleton consiste en :

1. L'établissement de GAC en $O(e_id^2 + \phi)$ sur le CN (ligne 3 de l'algorithme 3). Le terme $O(e_id^2)$ correspond à la propagation des e_i contraintes implicites binaires supplémentaires, à l'aide d'un algorithme d'AC optimal.
2. Traiter les $O(nd)$ modifications (valeurs supprimées), et supprimer les no-goods correspondants dans les contraintes implicites (lignes 8-12 de l'algorithme 3). L'exploitation des deltas des domaines modifiés par une propagation (comme suggéré par le schéma de propagation AC-5) permet de rendre ce travail incrémental, et la complexité amortie obtenue est en $O(e_id^2)$, qui peut être ignorée.

La seule structure de données utilisée par sDC-1 correspond au stockage des no-goods dans les contraintes binaires en extension, en $O(e_id^2)$. Les structures de données de l'algorithme d'AC optimal utilisé pour propager ces contraintes sont en $O(e_id)$ et peuvent être négligées. □

Les résultats expérimentaux décrits dans [11] indiquent que malgré la simplicité et la complexité théorique dans le pire des cas élevée, cet algorithme se comporte très bien en pratique, le plus souvent mieux que les algorithmes de PC état-de-l'art.

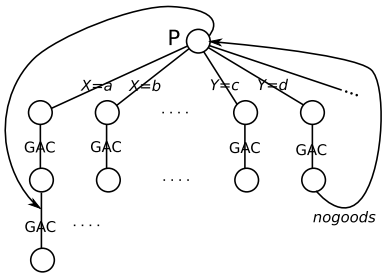


FIG. 1 – Illustration de l'incrémentalité de GAC sur les tests de singleton

2.2 sDC^{clone} : établir sDC en $O(ne_id^3 + nd\psi)$.

Cette complexité pour établir DC est obtenue en exploitant complètement l'incrémentalité des algorithmes de (G)AC. Comme illustré par la figure 1, si des no-goods sont déduits du test de singleton $Y = d$, le test de singleton antérieur $X = a$ ne devrait pas être recalculé entièrement, mais relancé incrémentalement à partir du dernier point fixe obtenu pour $X = a$. L'idée de sDC^{clone}, empruntée à l'algorithme SAC-OPT, est d'obtenir ce résultat en créant une copie physique du CN en mémoire pour chacun des $O(nd)$ tests de singleton possibles. Afin de bénéficier de l'incrémentalité des algorithmes de propagation, toutes les structures de données de ces algorithmes doivent également être dupliquées.

Il faut remarquer que les contraintes implicites doivent être propagées *à chaque fois qu'elles sont modifiées*, et non pas seulement quand une valeur est supprimée du domaine d'une variable. En consultant l'algorithme 1, on constate que chaque arc impliquant une contrainte implicite peut être inséré $O(d^2)$ fois, ce qui signifie $O(e_id^2)$ appels à `revise`. Bien que la procédure `revise` d'AC-2001 soit incrémentale, elle requiert au moins $O(d)$ opérations pour valider les supports courants. La complexité d'AC-2001 devient donc $O(e_id^3)$, et cet algorithme ne peut être utilisé pour obtenir une complexité optimale pour propager les contraintes implicites : il faut utiliser un algorithme à *grain fin* comme AC-4, AC-6 ou AC-7.

Proposition 2. Appliqué à un CN quelconque, sDC^{clone} admet une complexité temporelle dans le pire des cas en $O(e_ind^3 + nd\psi)$ et une complexité spatiale en $O(e_ind^2 + nd\rho)$.

Ébauche de preuve. La complexité temporelle dans le pire des cas est obtenue en considérant entièrement l'incrémentalité de GAC pour chacun des $O(nd)$ tests

de singleton, en supposant une complexité amortie pour établir GAC incrémentalement sur le CN en $O(\psi)$, et $O(e_i d^2)$ pour établir AC sur les contraintes implicites, en utilisant un algorithme d'AC optimal à grain fin.

Le CN doit être cloné $O(nd)$ fois. Cependant, la liste des no-goods (i.e. les relations des contraintes implicites) peut être partagé entre tous les clones. Seules les structures de données de l'algorithme d'AC optimal (en $O(e_i d)$) doivent être clonées. On en déduit la complexité spatiale de l'algorithme. □

Par exemple, pour un CN consistant uniquement en une série de contraintes *all-diff* ($O(\psi) = O(ek^2d^2)$ et $O(\rho) = O(ekd)$ [16]), par exemple pour modéliser un problème de carré latin, sDC peut être établie avec cet algorithme en $O(e_i nd^3 + enk^2d^3)$ avec une complexité spatiale en $O(e_i nd^2 + enkd^2)$.

Dans le cas binaire, sDC^{clone} atteint une complexité spatiale dans le pire des cas en $O(n(e + e_i)d^3)$, c'est-à-dire une complexité inférieure aux algorithmes de PC connus[5]. Cependant, les besoins en termes d'espace restent excessifs pour une utilisation pratique. De plus, Debruyne & Bessière indiquent dans [3] que les résultats expérimentaux conduits autour de l'algorithme SAC-OPT, construit suivant le même principe, se sont révélés décevants. L'algorithme n'est donc pas décrit ni expérimenté plus avant dans cet article.

2.3 sDC-2.1 : un compromis

« Restaurer l'état du CN, » une opération essentielle pour obtenir les complexités améliorées de sDC^{clone} ou SAC-OPT, peut être réalisé de manière très naturelle, sans aucune structure de données additionnelle, lors de l'établissement de DC. L'information enregistrée dans les contraintes implicites lors du précédent test du même singleton peut être exploitée pour restaurer les domaines des variables dans l'état dans lesquels ils étaient à la fin du précédent test. Ceci peut être effectué par un appel à la fonction `revise` des contraintes implicites impliquant la variable du singleton courant X_a : **pour ch.** $(C, Y) \in \text{mod}(\{X\}, \mathscr{C}_i)$ **faire** `revise`(C, Y). Ce processus est un *Forward Checking* et a une complexité dans le pire des cas en $O(nd)$, puisqu'il peut y avoir au plus n contraintes binaires impliquant une variable donnée, et que l'appel à `revise` sur une contrainte binaire dont l'une des variables est un singleton est en $O(d)$.

Cette observation avait été en partie exploitée pour concevoir l'algorithme sDC-2. Nous présentons ici sDC-2.1, une version optimisée et étendue de sDC-2, gérant les contraintes implicites et non-binaires de manière spécifique. L'algorithme est décrit par les algorithmes 4 et 5. La structure $lastModified$ de sDC-2 est remplacée par une nouvelle structure $arcQueue$ et par $O(nd)$ pointeurs (un par singleton X_a), notés $firstArc[X_a]$. Cette structure de données fonctionne comme suit : $arcQueue$ contient tous les arcs qui doivent être révisés lors de tous les tests de singleton. Un arc est inséré à la fin de la file lorsqu'une variable voisine ou la contrainte elle-même est modifiée. À chaque fois qu'un arc de $arcQueue$ est révisé pour le test de singleton X_a, $firstArc[X_a]$ est déplacé vers l'élément suivant de la file. Lors de l'appel à GAC, la file de propagation est initialisée avec tous les arcs de $arcQueue$ en commençant par $firstArc[X_a]$. Ce mécanisme assure que (1.) l'ajout d'un arc à la file de propagation se fait en temps constant tout en pouvant être géré par tous les singletons, et (2.) que les initialisations des files de propagation se font de manière incrémentale.

Ces structures de données sont utilisées à la ligne 5 de l'algorithme 4 et à la ligne 9 de l'algorithme 5 pour initialiser les files de propagation de l'algorithme sous-jacent, de sorte d'assurer que seuls les propagateurs susceptibles de réaliser des filtrages seront appelés (i.e. une variable impliquée par la contrainte ou la contrainte elle-même a été modifiée depuis le dernier test de ce singleton). X', Y', C' et $\mathscr{C}_i'$ sont les variables, contraintes ou ensembles de contraintes de P' qui correspondent respectivement à X, Y, C et $\mathscr{C}_i$ dans P.

Algorithm 4: sDC-2.1$(P = (\mathscr{X}, \mathscr{C}), \mathscr{C}_i)$

```
1  P ← GAC(P, mod(𝒳, 𝒞))
2  mark ← X ← first(𝒳)
3  répéter
4      si |dom(X)| > 1 ∧ checkVar-2.1(P, 𝒞_i, X)
       alors
5          P' ← GAC(P, {arcQueue[i] | firstArcMain ≤ i < cnt})
6          si P' = ⊥ alors retourner ⊥
7          pour ch. (Y, b) | Y ∈ 𝒳 ∧ b ∈ dom^P(Y) ∧ b ∉ dom^P'(Y) faire
8              retirer b de dom^P(Y)
9              pour ch. (C, Z) ∈ mod({Y}, 𝒞) faire
10                 arcQueue[cnt++] ← (C, Z)
11         firstArcMain ← cnt
12         mark ← X
13     X ← next(𝒳, X)
14 jusqu'à X = mark
15 retourner P
```

[5] Bien sûr, le nombre e_i de contraintes implicites ne peut être connu avant l'appel à l'algorithme, et peut atteindre $\binom{n}{2}$.

Algorithm 5: checkVar-2.1($P = (\mathscr{X}, \mathscr{C})$, $\mathscr{C}_i$, X)

```
1  modif ← faux
2  pour ch. a ∈ dom(X) faire
3      si cnt ≤ |𝒳| alors
           /* Premier tour                                   */
4          P' ← GAC(P|_{X=a}, mod({X}, 𝒞))
5      sinon
           /* Tours suivants : forward
              checking                                       */
6          P' ← P|_{X=a}
7          pour ch. (C', Y') ∈ mod({X'}, 𝒞'_i) faire
8              revise(C', Y')
           /* Propagation avec les files de
              propagation préinitialisées                    */
9          P' ←
           GAC(P', {arcQueue[i] | firstArc[X_a] ≤
           i < cnt}))
10     si P' = ⊥ alors
11         retirer a de dom^P(X)
12         pour ch. (C, X) ∈ mod({X}, 𝒞) faire
13             arcQueue[cnt++] ← (C, X)
14         modif ← vrai
15     sinon
16         pour ch. Y | {X, Y} ⊆ 𝒳 faire
17             soit C t.q. C ∈ 𝒞_i ∧ scp(C) = {X, Y}
18             pour ch. b ∈ dom^P(Y) | b ∉
               dom^{P'}(Y) ∧ {X_a, Y_b} ∈ rel(C) faire
19                 retirer {X_a, Y_b} de rel(C)
20                 arcQueue[cnt++] ← (C, X)
21                 arcQueue[cnt++] ← (C, Y)
22                 modif ← vrai
23 firstArc[X_a] ← cnt
24 retourner modif
```

Proposition 3. Appliqué à un réseau de contraintes quelconque, sDC-2.1 a une complexité temporelle dans le pire des cas en $O(e_i^2 d^4 + e_i n d^5 + nkd^2\phi)$ et une complexité spatiale en $O(e_i d^2 + ek^2 d)$.

Ébauche de preuve. Dans cette variante de l'algorithme, un test de singleton consiste en :

1. La restauration de l'état du CN à la fin du test de singleton précédent du même couple variable/valeur, en $O(nd)$ par Forward Checking. Le Forward Checking sur tous les singletons est amorti en $O(e_i d^2)$, et est réalisé $O(e_i d^2)$ fois dans le pire des cas, s'agissant du nombre de modifications possibles. D'où un premier terme en $O(e_i^2 d^4)$.
2. L'initialisation de la file de propagation de GAC. Dans le pire des cas, chacun des $O(e_i + ek)$ arcs est inséré une fois à chaque modification possible de chaque test de singleton ($O(d^2)$ pour les arcs impliquant les contraintes implicites et $O(kd)$ pour les autres), d'où une complexité amortie en $O(nd.(e_i d^2 + ek^2 d))$.
3. La propagation de l'assignation ou des mises à jour. Puisque les structures de données des propagateurs ne sont pas clonées, l'incrémentalité de ceux-ci est perdue, et retombe à $O(e_i d^2 + \phi)$. On sait qu'un propagateur donné ne peut être appelé que $O(kd)$ fois pour un singleton donné (resp. $O(d^2)$ fois pour les contraintes implicites), quand une valeur (resp. un no-good) est supprimée. Pour chacun des $O(nd)$ singletons, la complexité amortie pour toutes les propagations d'un test de singleton est donc en $O(e_i d^4 + kd\phi)$.
4. Si une inconsistance est détectée, la suppression d'une valeur et la mise à jour de *arcQueue* (ligne 13 de l'algorithme 5). La détection d'une inconsistance à la ligne 10 ne peut survenir que $O(nd)$ fois et nécessite $O(n)$ opérations pour la mise à jour d'*arcQueue*, qui sont amorties pour un total en $O(e_i d + ed)$. On peut négliger ce terme en considérant $e \leq \phi$.
5. Sinon, la mise à jour des contraintes implicites et de *arcQueue*. Comme pour sDC-1, ces opérations sont incrémentales si les deltas sont exploités, et peuvent être négligées.

En ce qui concerne la complexité spatiale : les relations des contraintes implicites sont en $O(e_i d^2)$. Les $O(e_i)$ arcs correspondant aux contraintes implicites peuvent être insérés $O(d^2)$ fois dans la file de révision, et les $O(ek)$ arcs correspondant aux contraintes initiales peuvent être insérées $O(kd)$ fois. La structure *arcQueue* a donc une complexité spatiale en $O(e_i d^2 + ek^2 d)$ et les pointeurs $firstArc$ en $O(nd)$. Ce dernier terme peut être négligé, en considérant $\rho \geq nd$. □

Appliqué à un graphe complet de contraintes binaires ($O(\phi) = O(n^2 d^2)$ et $O(e_i) = O(n^2)$), la complexité de l'algorithme devient $O(n^4 d^4 + n^3 d^5)$, ce qui présente une amélioration par rapport à sDC-2 (en $O(n^5 d^5)$). Si l'on considère par exemple un CN composé initialement de contraintes *all-diff*, pouvant être propagées en $O(k^{1.5} d^2)$ [16], on peut ainsi établir sDC en $O(e_i^2 d^4 + e_i n d^5 + enk^{2.5} d^4)$.

2.3.1 Notes d'implémentation

En pratique, la structure *arcQueue* peut sans risque être remplacée par deux tableaux d'entiers

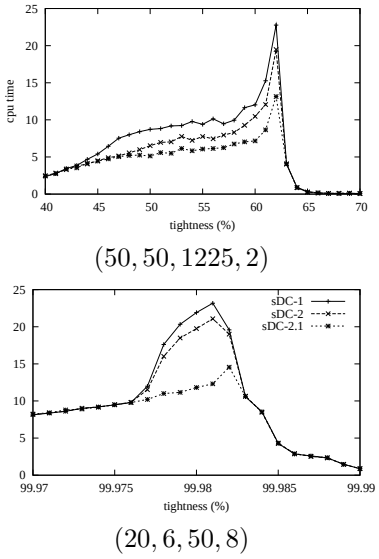


FIG. 2 – Prétraitement sur des instances aléatoires présentant des contraintes de dureté variable. Temps CPU en secondes. Les caractéristiques des problèmes sont données sous forme de quadruplets (n, d, e, k).

$lastModVar$ et $lastModCons$, c'est-à-dire une structure plus proche de l'algorithme sDC-2 original. L'initialisation des files de propagation avant chaque propagation n'est alors plus incrémentale et ajoute un terme théorique en $O(e_i n^2 d^3 + e_i^2 n d^3)$ à la complexité temporelle dans le pire des cas de l'algorithme. Cependant, la complexité spatiale est moindre ($O(n + e_i)$ au lieu de $O(nd + e_i d^2 + ekd)$), et le résultat est plus rapide en pratique, cette structure permettant de regrouper naturellement plusieurs révisions d'un même arc.

L'algorithme de GAC « orienté arcs » présenté à l'algorithme 1 peut être émulé par une version « orientée variables » améliorée, exploitant des heuristiques d'ordonnancement des révisions et des compteurs spéciaux pour éviter les révisions inutiles, comme décrit dans [6, 7].

3 Expérimentations

Les expérimentations sont effectuées sur des *benchmarks* de la Troisième Compétition Internationale de Solveurs [18], sur des problèmes binaires et non-binaires, impliquant des contraintes en intention, extension ou présentant des propagateurs spécifiques connus. Nous avons utilisé le solveur Concrete, construit à partir de la bibliothèque de programmation CSP4J [20].

La figure 2 donne une idée de l'amélioration apportée par sDC-2.1 par rapport à sDC-1 et sDC-2, même sur des problèmes binaires. sDC est établie sur des problèmes binaires (graphe du haut) et non-binaires (graphe du bas) aléatoires, avec chacun des algorithmes. Les différences entre les algorithmes apparaissent près du point de transition, quand les singletons doivent être parcourus plusieurs fois pour atteindre le point fixe.

Deux stratégies sont comparées : les problèmes sont résolus en utilisant un algorithme MGAC-*dom/wdeg* classique, après une phase de préparation où GAC ou sDC sont respectivement établis. Les structures de données et les algorithmes de propagation utilisés pour les contraintes binaires en extension sont ceux d'AC-3^{bit} ou AC-3^{bit+rm}.

La table 1 montre des résultats représentatifs d'instances pour lesquelles l'application de sDC a un impact positif sur la recherche. sDC est particulièrement efficace sur les problèmes très difficiles, où le nombre réduit de nœuds explorés parvient à compenser le temps passé durant le prétraitement et celui passé à propager les contraintes supplémentaires pendant la recherche (ce temps peut être estimé par la métrique *nœuds par seconde*). Les problèmes sélectionnés sont des problèmes structurées présentant différents types de contraintes. Les instances *taillard* sont des problèmes d'ordonnancement d'atelier, modélisés par des contraintes d'*inégalité disjonctive* avec un propagateur incrémental spécifique à complexité linéaire. *tsp* est un problème de voyageur de commerce et *series* un problème de séries d'intervalles impliquant des contraintes de tables positives ternaires, propagées avec un algorithme STR [17]. *scen11-f1* est l'instance de problème d'allocation de fréquences RFLAP la plus difficile. Bien que cette instance n'implique que des contraintes binaires, elle montre qu'il n'est pas nécessaire de compléter le graphe de contraintes pour établir DC (*scen11-f1* utilise 680 variables et compléter le graphe nécessiterait plus de 230 000 contraintes).

Dans de nombreux cas, les nouvelles contraintes interfèrent avec l'heuristique de choix de variable : les meilleures heuristiques de choix de variable génériques, comme *dom/wdeg*, exploitent en effet la structure du CN pour sélectionner les variables à affecter. L'instance *tsp-25-715* est influencée de manière spectaculaire par ce phénomène.

La table 2 montre des exemples d'instances pour lesquelles l'application de sDC est contre-productive. Trois principaux inconvénients peuvent être identifiés, et une instance représentative de chaque cas a été choise : pour *os-taillard-7-95-7*, bien que le temps de prétraitement soit très raisonnable, la réduction du nombre de nœuds ne permet pas de compenser le temps perdu à propager les nouvelles contraintes. Dans le cas de *os-gp-10-10*, un très grand nombre de contraintes implicites est généré (jusqu'à 5 000 pour

		GAC	sDC	GAC	sDC
		tsp-25-715		*scen11-f1*	
		$(76, 1\,001, 350, 3)$		$(680, 42, 4\,103, 2)$	
prepro	cpu	0	353	0	41
	add cstr	0	2 303	0	757
search	cpu	497	**23**	9 646	**9 408**
	nodes	397 k	**2k**	6 749 k	**4 335 k**
	nodes/s	**798**	96	**700**	461
total	cpu	497	**376**	9 646	**9 448**
		series-15		*os-taillard-7-100-0*	
		$(29, 15, 210, 3)$		$(49, 434, 294, 2)$	
prepro	cpu	0	1	0	27
	add cstr	0	182	0	294
search	cpu	475	**238**	> 600	**104**
	nodes	6 920 k	**1 890 k**	$> 2\,790$ k	**148 k**
	nodes/s	**15 142**	7 908	**4 650**	1 423
total	cpu	475	**239**	> 600	**131**

Tab. 1 – Comparaison des performances de la recherche avec et sans prétraitement par sDC. Instances positives représentatives.

		GAC	sDC	GAC	sDC	GAC	sDC
		os-taillard-7-95-7		*os-gp-10-10-1092*		*graceful-K5-P2*	
		$(49, 403, 294, 2)$		$(101, 1\,090, 1\,000, 2)$		$(35, 26, 370, 3)$	
prepro	cpu	0	25	0	> 275	0	5
	add cstr	0	303	0	$> 1\,020$	0	200
search	cpu	**675**	1,654	$> 2\,400$	–	**157**	246
	assgn	2 749 k	**1,698 k**	> 864 k	–	**985 k**	993 k
	nodes/s	**4 073**	1,026	360	–	**6 280**	4 040
total	cpu	**675**	1,679	$> 2\,400$	–	**157**	251

Tab. 2 – Instances représentatives négatives.

des tailles de domaines de l'ordre de 1 000 valeurs), et elles ne peuvent alors plus être stockées en mémoire (dans une limite de 600 Mio). Finalement, dans le cas de l'instance de *graceful*, *dom/wdeg* est perturbé par les nouvelles contraintes. Nous avons cependant constaté que ce cas reste assez rare, et d'autres stratégies de recherches plus proches de la sémantique des problèmes ne seraient bien sûr pas affectées.

		GAC	sDC/RC
		os-gp-10-10-1092	
prepro	cpu	799	
	add cstr	851	
search	cpu	599	
	nodes	321k	
	nodes/s	536	
total	cpu	1,398	

Tab. 3 – Résoudre un problème en générant des contraintes convexes par rangée implicites

4 Perspectives

Le principal intérêt de la consistance duale est de pouvoir automatiquement améliorer la robustesse des solveurs face à des modèles naïvement modélisés. En effet, certaines des contraintes implicites introduites par DC pourraient être insérées manuellement pendant la phase de modélisation, avec un algorithme de propagation approprié et des structures de données plus légères.

Cependant, comme les contraintes ajoutées par la consistance duale sont des contraintes implicites, elles peuvent être facilement relâchées sans ajouter de solutions au CSP. La DC conservative proposée dans [10] est déjà un bon exemple d'une telle approximation : les no-goods qui ne peuvent être ajoutés aux contraintes originellement présentes dans le problème sont ignorés. En perspective à nos travaux, nous proposons ces extensions :

– Relâcher les contraintes implicites pour qu'elles respectent une certaine propriété, comme la convexité par rangées (RC). De telles propriétés permettent de propager plus efficacement les contraintes implicites. La table 3 donne un

exemple de problème difficile résolu par l'ajout de contraintes convexes RC grâce à DC (les nogoods qui violent la propriété sont ignorés). La DC complète n'avait pas pu être établie sur le même problème (cf table 2).
– Ne propager les contraintes implicites que si elles sont actives pendant la recherche. On pourra se référer à des travaux tels que [15] pour cela, sans avoir à détecter les contraintes redondantes.

5 Conclusion

Dans cet article, nous avons proposé une extension de la définition de la consistance duale, et donc de la consistance de chemin, aux réseaux de contraintes non-binaires. Nous avons montré comment utiliser DC pour déduire automatiquement des contraintes binaires implicites à partir des domaines et contraintes originaux du CN. Une nouvelle variante d'algorithme de consistance duale forte, nommée sDC^{clone}, présentant une complexité dans le pire des cas intéressante, a été décrite. Un compromis efficace, gérant spécifiquement contraintes non-binaires et contraintes implicites, nommé sDC-2.1, a été proposé. Des expérimentations de ces algorithmes, utilisant la génération de contraintes implicites « à la volée » ont été décrites, et montré combien cette approche était prometteuse.

Les expérimentations ont également permis de mettre en évidence les faiblesses potentielles de la consistance duale, pouvant empêcher son utilisation sur des problèmes industriels. De nouvelles idées prospectives ont été proposées pour apporter des solutions à ces inconvénients.

Remerciements. Ces travaux ont été réalisés dans le cadre du projet CANAR (ANR-06-BLAN-0383-02). L'auteur tient à remercier Christian Bessière, Thierry Petit, les autres membres des projets CANAR et COCONUT, ainsi que les relecteurs anonymes pour leur aide.

Références

[1] N. Beldiceanu and E. Contejean. Introducing global constraints in CHIP. *Mathl. Comput. Modelling*, 20(12) :97–123, 1994.

[2] C. Bessière. *Handbook of Constraint Programming*, chapter 3 : Constraint Propagation, pages 29–84. Elsevier, 2006.

[3] C. Bessière and R. Debruyne. Theoretical analysis of singleton arc consistency and its extensions. *Artificial Intelligence*, 172(1) :29–41, 2008.

[4] C. Bessière and J.-C. Régin. Arc consistency for general constraint networks : preliminary results. In *Proceedings of IJCAI'97*, 1997.

[5] C. Bessière, J.-C. Régin, R.H.C. Yap, and Y. Zhang. An optimal coarse-grained arc consistency algorithm. *Artificial Intelligence*, 165(2) :165–185, 2005.

[6] F. Boussemart, F. Hemery, and C. Lecoutre. Revision ordering heuristics for the Constraint Satisfaction Problem. In *Proceedings of CPAI'04 workshop held with CP'04*, pages 29–43, 2004.

[7] F. Boussemart, F. Hemery, C. Lecoutre, and L. Saïs. Support inference for generic filtering. In *Proceedings of CP'04*, pages 721–725, 2004.

[8] R. Dechter. *Constraint processing*. Morgan Kaufmann, 2003.

[9] I.P. Gent, C. Jefferson, and I. Miguel. Watched literals for constraint propagation in Minion. In *Proceedings of CP'06*, pages 182–197, 2006.

[10] C. Lecoutre, S. Cardon, and J. Vion. Conservative Dual Consistency. In *Proceedings of AAAI'07*, pages 237–242, 2007.

[11] C. Lecoutre, S. Cardon, and J. Vion. Path Consistency by Dual Consistency. In *Proceedings of CP'2007*, 2007.

[12] C. Lecoutre and F. Hemery. A study of residual supports in arc consistency. In *Proceedings of IJCAI'2007*, pages 125–130, 2007.

[13] C. Lecoutre and J. Vion. Enforcing Arc Consistency using Bitwise Operations. *Constraint Programming Letters*, 2 :21–35, 2008.

[14] U. Montanari. Network of constraints : Fundamental properties and applications to picture processing. *Information Science*, 7 :95–132, 1974.

[15] C. Piette. Let the solver deal with redundancy. In *Proceedings of ICTAI'08*, volume 1, pages 67–73, 2008.

[16] J.-C. Régin. A filtering algorithm for constraints of difference in CSPs. In *Proceedings of AAAI'94*, pages 362–367, 1994.

[17] J.R. Ullmann. Partition search for non-binary constraint satisfaction. *Information Science*, 177 :3639–3678, 2007.

[18] M. van Dongen, C. Lecoutre, and O. Roussel. Third International CSP Solvers Competition. http ://www.cril.univ-artois.fr/CPAI08, 2008.

[19] P. van Hentenryck, Y. Deville, and CM. Teng. A generic arc-consistency algorithm and its specializations. *Artificial Intelligence*, 57 :291–321, 1992.

[20] J. Vion. Constraint Satisfaction Problem for Java. http ://cspfj.sourceforge.net/, 2006.

Utilisation de sous-systèmes bien contraints pour le filtrage de CSP numériques

Ignacio Araya, Gilles Trombettoni, Bertrand Neveu
INRIA, Université de Nice-Sophia, CERTIS

{Ignacio.Araya,neveu,trombe}@sophia.inria.fr

Résumé

Quand les méthodes par intervalles sont appliquées aux systèmes d'équations sur les réels, deux principaux types de filtrage sont utilisés pour réduire l'espace de recherche. Quand le système est carré, l'algorithme de Newton sur intervalles se comporte comme une contrainte globale sur tout le système $n \times n$. D'autre part, les algorithmes de filtrage, provenant de la programmation par contraintes, effectuent une boucle de propagation de type AC3, où les contraintes sont traitées itérativement *une par une*.

Cet article étudie la possibilité de filtrer des *sous-systèmes bien contraints* $k \times k$, avec $1 \leq k \leq n$. Nous donnons les définitions théoriques des consistances partielles *Box-k-consistance* et *kB-consistance structurelle* qui généralisent la box-consistance. Nous montrons que les sous-systèmes bien contraints se comportent comme des contraintes globales $k \times k$ qui peuvent apporter un filtrage supplémentaire par rapport à Newton sur intervalles et aux sous-systèmes 1×1 standard. La contraction obtenue pendant le processus de résolution des sous-systèmes permet également d'apprendre des points de choix multi-dimensionnels, c.-à-d. de bissecter simultanément plusieurs domaines de variables dans l'arbre de recherche. Des expérimentations soulignent les gains en temps CPU obtenus sur des systèmes décomposés et structurés par rapport aux algorithmes connus.

1 Introduction

Quand les méthodes par intervalles sont appliquées aux systèmes d'équations sur les réels, deux principaux types de filtrage sont utilisés pour réduire l'espace de recherche. Quand le système est carré , c.-à-d., contient n variables contraintes par n équations, l'algorithme de Newton sur intervalles (`I-Newton`) se comporte comme une contrainte globale sur une linéarisation du système entier $n \times n$. Les algorithmes de filtrage, provenant de la programmation par contraintes, effectuent une boucle de propagation de type AC3, où les contraintes sont traitées itérativement une par une par une procédure *revise* qui, comme par exemple `BoxNarrow`, filtre un sous-système 1×1.

Cet article étudie la possibilité de filtrer des sous-systèmes bien contraints $k \times k$, avec $1 \leq k \leq n$. Nous définissons de nouvelles consistances partielles sur des sous-systèmes qui sont bien contraints, ce qui ajoute une restriction structurelle à la kB-consistance [13]. La propriété d'être structurellement bien contraint ainsi que les bases sur les techniques à intervalles sont définies dans la partie 2. La Box-k-consistance et la S-kB-consistance présentées dans la partie 6 s'avèrent être une généralisation de la Box-consistance [2].

Nous détaillons dans la partie 4 les procédures *revise* qui filtrent un sous-système pour le rendre Box-k ou S-kB consistant. Cette procédure développe un arbre de recherche local, à l'intérieur du sous-système et utilise un algorithme `I-Newton` local. Ces procédures *revise* ont des points communs avec l'algorithme proposé dans [7]. Leur algorithme réalise aussi une recherche arborescente où chaque nœud est filtré avant de retourner une approximation extérieure des sous-boîtes obtenues. Mais il est appliqué au système d'équations entier et non à des sous-systèmes bien contraints.

La partie 5 présente en détail comment les arbres de recherche locaux construits dans les sous-systèmes permettent à une stratégie de résolution d'apprendre des points de choix multidimensionnels dans l'arbre de recherche global, c.-à-d. de couper les domaines de plusieurs variables simultanément. Ces points de choix multidimensionnels sont appelés *multisplits*. Des expérimentations prometteuses montrent l'apport de notre approche pour des systèmes de contraintes numériques (NCSP) décomposés et structurés qui sont fréquents

en pratique. Nous montrons aussi que le schéma filtrage S-kB + multisplit généralise l'algorithme de retour arrière entre blocs (`IBB`) [17, 16] dédié à la résolution de systèmes décomposés (comme par exemple certains systèmes de contraintes géométriques).

2 Fondements

Les algorithmes présentés dans cet article ont pour but de résoudre des systèmes d'équations, ou plus généralement des CSP numériques.

Définition 1 *Un* **CSP numérique** *(NCSP)* $P = (X, C, B)$ *comprend un ensemble* X *de* n *variables et un ensemble* C *de contraintes sur ces variables. Chaque variable* $x_i \in X$ *peut prendre une valeur réelle dans l'intervalle* $[x_i]$ *et* B *est le produit cartésien (appelé* **boîte***)* $[x_1] \times ... \times [x_n]$*. Une solution de* P *est une affectation des variables de* X *satisfaisant toutes les contraintes de* C*.*

Comme les ordinateurs ne savent pas représenter les nombres réels, les bornes d'un intervalle $[x_i]$ doivent être définis comme des nombres à virgule flottante. Les opérations ensemblistes comme l'inclusion et l'intersection sont définis sur des boîtes. Un opérateur d'enveloppe `Hull` est souvent utilisé pour calculer une approximation englobante de l'union de plusieurs boîtes.

Définition 2 *Soit* $S = \{[b_1], ..., [b_n]\}$ *un ensemble de boîtes sur les mêmes variables.*
Nous appelons enveloppe de S*, notée* `Hull`*(*S*), la boîte minimale incluant* $[b_1], [b_2], ..., [b_n]$*.*

Certains NCSP admettent un nombre fini de solutions. Ce sont généralement des systèmes $n \times n$ d'équations indépendantes. Ils sont **carrés** dans le sens que n équations contraignent n variables. Le graphe de contraintes biparti correspondant, dont les sommets sont les variables d'une part et les contraintes d'autre part, vérifie la propriété structurelle suivante.

Définition 3 *Un NCSP est* **(structurellement) bien-contraint** *si son graphe biparti correspondant admet un couplage parfait [9].*

Pour trouver toutes les solutions d'un NCSP avec des techniques par intervalles, le processus de résolution commence avec une boîte initiale représentant l'espace de recherche. Il construit un arbre de recherche en bissectant la boîte courante, c.-à-d. en la coupant en deux sur une dimension (une variable), créant deux sous-boîtes. A chaque nœud de l'arbre de recherche, des algorithmes de filtrage (ou contraction) réduisent la boîte courante. Ces algorithmes comprennent des algorithmes `I-Newton` provenant de la communauté d'analyse numérique [10, 15] et des algorithmes de contraction provenant de la communauté de programmation par contraintes. Le processus s'arrête quand on a obtenu des **boîtes atomiques** de taille inférieure à la précision demandée ϵ sur chaque dimension.

Le nouvel algorithme de contraction présenté dans cet article généralise l'algorithme bien connu `Box` qui établit la propriété de *Box-consistance* [2] définie ainsi :

Définition 4 *Un NCSP* (X, C, B) *est* **box-consistant** *si chaque paire* (c, x) *est box-consistante (*$c \in C$*,* $x \in X$ *et* x *est une des variables contraintes par* c*).*

Considérons une paire (c, x)*, où* $c(x, y_1, ..., y_a) = 0$ *est une équation d'arité* $a+1$*. Soit* c' *l'équation* c *où les variables* y_i *sont remplacées par leur domaine courant dans* B *:* $c'(x) = c(x, [y_1], ..., [y_a]) = 0$*.* (c, x) *est box-consistant si :*

- $0 \in c'([\underline{[x]}, +]) = c([\underline{[x]}, +], [y_1], ..., [y_a])$ *;*
- $0 \in c'([-, \overline{[x]}]) = c([-, \overline{[x]}], [y_1], ..., [y_a])$*.*

$\underline{[x]}$*, resp.* $\overline{[x]}$*, est la borne inférieure, resp. supérieure, de* $[x]$*.* $[\underline{[x]}, +]$ *est l'intervalle étroit (d'un u.l.p.) dont les bornes sont* $\underline{[x]}$ *et le nombre flottant suivant.* $[-, \overline{[x]}]$ *est l'intervalle dont les bornes sont le nombre précédant* $\overline{[x]}$ *et* $\overline{[x]}$*.* c *représente indifféremment l'expression analytique sur les réels et la fonction étendue aux intervalles.*

En pratique, l'algorithme `Box` réalise une propagation de type AC3. Pour chaque paire (c, x), il réduit les bornes de $[x]$ de telle sorte que la nouvelle borne inférieure (resp. supérieure) soit la plus petite solution (resp. la plus grande) de l'équation à une variable $c'(x) = 0$. Les procédures *revise* existantes utilisent un principe de *rognage* pour réduire $[x]$. Des tranches $[s_i]$ dans $[x]$ sont enlevées si $c([s_i], [y_1], ..., [y_a])$ ne contient pas 0 (on utilise pour cela `I-Newton` univarié).

Une autre consistance partielle bien connue est la kB-consistance [13].

Définition 5 *Une contrainte* c *d'un NCSP* (X, C, B) *est* **2B-consistante** *(ou hull-consistante) si la boîte* B *est la plus petite boîte englobant toutes les solutions de* c*. Un NCSP* P *est* **kB-consistant** *si, pour toute variable* x*, les fermetures par* $(k-1)B$*-consistance de* $P_{\underline{[x]}}$ *(c.-à-d.,* P *avec* $[x]$ *remplacé par* $\underline{[x]}$*) et de* $P_{\overline{[x]}}$ *ne sont pas vides.*

L'algorithme `HC4` [2] calcule en gros la 2B-consistance d'un NCSP **ternarisé** dans lequel on a introduit des variables auxiliaires (et des équations additionnelles) pour faire disparaître les occurrences

multiples des variables. La 2B-consistance et la kB-consistance sont similaires à resp. l'arc-consistance et la consistance SAC [8] restreintes aux bornes des intervalles.

3 Consistances partielles Box-k et S-kB

Consistance Box-k

Comme nous l'avons expliqué auparavant, la Box-consistance produit une boîte englobante pour des sous-systèmes 1×1 (c, x). La Box-k-consistance introduite dans cet article généralise la Box-consistance en produisant une approximation extérieure de sous-systèmes $k \times k$ bien contraints. La Box-k-consistance se focalise sur des sous-systèmes suffisamment denses pour pouvoir réduire fortement l'espace de recherche.

Définition 6 *Soit $P' = (X', C', B')$ un sous-système du NCSP $P = (X, C, B)$ ($|X'| = |C'| = k$), obtenu par remplacement des* **variables d'entrée** *(les variables présentes dans au moins une contrainte de C' mais n'appartenant pas à X') par leur intervalle courant dans B. Les contraintes C' sont des équations et le sous-graphe (X', C') est connexe et structurellement* **bien contraint**, *c.-à-d., admet un couplage parfait.*

Le sous-système P' est **Box-k-consistant** *si il existe une boîte de taille 1 u.l.p. sur chaque face de la k-boîte B' telle que toute contrainte c dans C' est "satisfaite", c.-à-d., $0 \in c(X')$.*

Un NCSP est Box-k-consistant si tous ses sous-systèmes bien contraints de taille inférieure ou égale à k sont Box-k-consistants.

La figure 1-gauche montre un exemple de sous-système 2×2. La boîte extérieure est Box-consistante puisqu'elle approxime chaque contrainte de manière optimale. La boîte intérieure est Box-2-consistante puisqu'elle approxime de manière optimale l'ensemble des 6 solutions *épaisses* des deux contraintes. Les contraintes sont *épaisses* car les variables d'entrée (ex : w_1, w_2, w_3) sont remplacées par des intervalles. La figure 1-droite montre le couplage parfait (arêtes en gras) du sous-graphe correspondant.

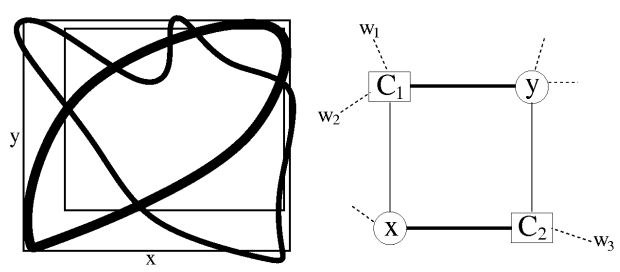


FIG. 1 – Illustration de la Box-2-consistance

Remarques

Pour des raisons d'efficacité, les consistances partielles des NCSP sont en pratique définies avec une précision ϵ, qui doit alors remplacer 1 u.l.p. dans les définitions précédentes.

Restreindre les sous-systèmes à des sous-graphes d'équations bien contraints a deux avantages. D'abord, cela permet un filtrage fort dans des sous-graphes spécifiques, ce qui est utile pour des NCSP peu denses ou pour des systèmes (globalement) sous-contraints, par exemple des systèmes mélangeant équations et inégalités. Ensuite, cela permet d'utiliser `I-Newton` pour contracter le sous-système. Comme la complexité en temps de la méthode `I-Newton` est cubique dans le nombre de variables, son application peut être trop chère pour de très grands NCSP. C'est pourquoi il est intéressant dans ce cas d'utiliser `I-Newton` uniquement pour des sous-systèmes.

Consistance S-kB

Cette nouvelle consistance partielle est une restriction de la bien connue kB-consistance, ou plus précisément $(k+2)$B-consistance, pour laquelle seuls les sous-systèmes bien contraints de taille k sont considérés. Elle est plus forte que la Box-k-consistance car non seulement les k-points sont consistants pour les contraintes à l'intérieur d'un sous-système $k \times k$ (comme Box-k), mais aussi les k-points sont 2B-consistants vis à vis du système entier.

Définition 7 *Soit $P' = (X', C', B')$ un sous-système du NCSP $P = (X, C, B)$ ($|X'| = |C'| = k$), obtenu par remplacement des* **variables d'entrée** *par leur intervalle courant dans B. Les contraintes dans C' sont des équations ; le sous-graphe (X', C') est connexe et structurellement bien-contraint.*

Le sous-système P' est **structurellement kB-consistant** *(en abrégé :* **S-kB-consistant***) si il existe une boîte $B\epsilon$ de taille 1 u.l.p. sur chaque face de la k-boîte B' telle que :*

- *toute contrainte c de C' est "satisfaite" : $0 \in c(X')$, et*
- *le système entier $(X, C, B\epsilon)$ est 2B-consistent.*

Il est bien connu que la 3B-consistance est plus forte que la Box-consistance [6], même si la 3B-consistance est calculée sur le système ternarisé, c.-à-d., si l'algorithme `HC4` est utilisé pour enlever des tranches dans le processus de rognage. Ce théorème peut être immédiatement généralisé comme suit.

Proposition 1 *La S-kB-consistance est plus forte que la Box-k-consistance. La $(k+2)B$-consistance est plus forte que la S-kB-consistance ($k \geq 1$).*

Pour comprendre pourquoi la S-kB-consistance (définition 7) est liée à la (k+2)-B-consistance, nous devons "déplier" la définition (5) récursive de cette dernière.

Apports de ces consistances structurelles partielles

L'exemple suivant montre comment une contraction obtenue sur un sous-système $k \times k$ peut être plus forte que sur des sous-systèmes 1×1 (2B-consistance) et sur le système entier $n \times n$ réalisé par `I-Newton`. Considérons le NCSP $P = (\{x, y, z\},\ \{x - y = 0,\ x + y + z = 0,\ (z-1)(z-4)(2x+y+2) = 0\},\ \{[-10^6, 10^6], [-10^6, 10^6][-10, 10]\})$.

Les contracteurs `HC4` (ou `Box`) et `I-Newton` sur P ne filtrent pas la boîte. Le calcul de la Box-2-consistance sur le sous-système 2×2 ({x,y}, {x-y=0, x+y+z=0}) réduit les intervalles de x et y à $[-5, 5]$ comme on le voit sur la figure 2. De plus, si des étapes de bissection sont utilisées pour trouver les solutions, il ne faut que deux points de choix pour isoler les 3 solutions $\{(\frac{-2}{3}, \frac{-2}{3}, \frac{4}{3}), (-0.5, -0.5, 1), (-2, -2, 4)\}$. Soulignons que `I-Newton` sur le système entier ne contracte pas la boîte car elle contient plusieurs solutions, tandis que `I-Newton` sur le sous-système 2×2 la contracte, car elle ne contient qu'une solution (épaisse en z) : le segment en gras. Bien sûr, ce petit exemple est seulement présenté à des fins didactiques. Les expérimentations décrites dans la partie 6 montrent sur de plus grandes instances non linéaires les gains de ces consistances partielles par rapport à d'autres consistances partielles comme la 3B-consistance [13].

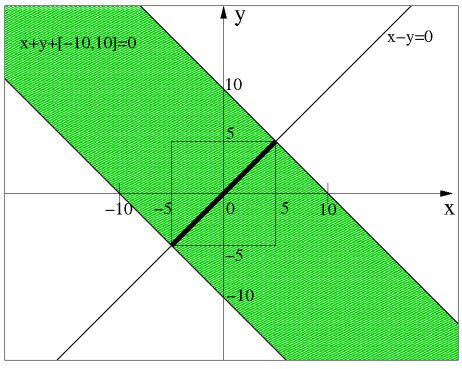


FIG. 2 – Illustration du sous-système de taille 2 avec $z = [-10, 10]$ comme variable d'entrée. {[-5,5],[-5,5]} est box-2-consistant pour le (sous-)système constitué des 2 contraintes $x - y = 0$ et $x + y + [z] = 0$.

4 Algorithmes de contraction utilisant des sous-systèmes bien contraints comme contraintes globales

Réaliser ces consistances partielles structurelles sur tous les sous-systèmes d'une taille donnée k est trop cher et contre-productif en pratique. Le nombre de sous-systèmes de taille k est grand et cet aspect exponentiel demeure même si on se limite aux systèmes bien contraints [3].

C'est pourquoi nous avons conçu une propagation de type AC3 qui gère des sous-systèmes de différentes tailles : des sous-systèmes de taille 1 mais aussi des sous systèmes bien contraints plus grands. Ces sous-systèmes sont ainsi similaires à des contraintes globales [18, 12] qui peuvent être définies par l'utilisateur ou automatiquement (voir partie 6).

Tous les sous-systèmes sont d'abord mis dans une queue de propagation et contractés en séquence. Quand le domaine d'une variable est réduit d'un ratio plus grand que ρ_{propag}, tous les sous-systèmes contenant cette variable sont ajoutés dans la queue, s'ils n'y sont pas déjà. Cette propagation est spécialisée par la procédure *revise* utilisée pour contracter les sous-systèmes de taille supérieure à 1 et détaillée ci après.

4.1 Les procédures `Box-k-revise` et `S-kB-revise`

La procédure *revise* est basée sur une méthode *branch & prune*, qui limite la bissection aux k variables de sortie X du sous-système, et utilise une stratégie de recherche en *largeur d'abord*. A la fin du parcours de cet arbre de recherche local, la boîte courante est remplacée par la boîte enveloppe des feuilles de l'arbre local. L'algorithme `Boxk-SkB-Revise` est une procédure générique qui calcule la Box-k-consistance du sous-système si le paramètre C contient les k équations du sous-système alors qu'elle effectue la S-kB-consistance du sous-système si C contient toutes les équations du NCSP. Cette procédure gère une liste L de nœuds, qui sont les feuilles de l'arbre local. Une feuille l de L a 3 attributs principaux : $l.box$ désigne l'espace de recherche (n-dimensionnel) associé au nœud ; $l.precise$ est un booléen indiquant si $l.box$ a atteint la précision ϵ sur toutes les dimensions (ϵ est aussi la précision requise pour la solution globale) ; $l.certified$ est un booléen indiquant si $l.box$ contient une solution unique. Le paramètre `box` est la boîte globale courante (espace de recherche) quand la procédure *revise* est appelée.

Un processus combinatoire (recherche arborescente) est réalisé par la boucle `while`. A chaque itération, une feuille de L, qui n'est ni *precise* ni *certified*, est choisie, bissectée et les deux nouvelles sous-boîtes sont contractées. La recherche s'arrête quand toutes les feuilles sont

Algorithm 1 `Boxk-SkB-Revise` (**in-out** L, `box`; **in** X, C, ϵ, `subContractor`, τ_{leaves}, $\tau_{\rho_{io}}$)

```
UpdateLocalTree(L, box, X, C, ε, subContractor)
L' ← {l ∈ L s.t. ¬l.certified and ¬l.precise and
ProcessLeaf(l,X,C,τ_ρio)}
while 0 < L'.size and L.size < τ_leaves do
  l ← L'.front() /* Choisit une feuille en largeur d'abord */
  (l1, l2) ←bisect(l, X)
  contract(l1, subContractor, X, C, ε)
  contract(l2, subContractor, X, C, ε)
  if l1.box ≠ ∅ then L.pushBack(l1) end if
  if l2.box ≠ ∅ then L.pushBack(l2) end if
  L.remove(l)
  L' ← {l ∈ L s.t. ¬l.certified and ¬l.precise
  and ProcessLeaf(l,X,C,τ_ρio)}
end while
box ←hull(L) /* Enveloppe de l'union de toutes les boîtes
l.box, l ∈ L */
```

étiquetées comme *certified* ou *precise* ou quand on a atteint une limite τ_{leaves} (fixée expérimentalement à 10) pour le nombre de feuilles. τ_{leaves} limite les besoins en mémoire (voir partie 4.5) et permet de propager rapidement les réductions obtenues aux autres sous-systèmes.

Une feuille est simplement choisie en largeur d'abord. Nous avions d'abord essayé une heuristique plus sophistiquée pour choisir une grosse boîte sur le bord de l'enveloppe des différentes feuilles. L'idée était de maximiser le gain en volume sur la boîte globale courante au cas où la feuille sélectionnée serait éliminée par filtrage. Nous avons écarté cette généralisation multi-dimensionnelle de l'algorithme `BoxNarrow` (qui rogne les bornes de l'intervalle traité dans l'algorithme de `Box`) car elle n'apportait pas de gains de performance significatifs.

Algorithm 2 `contract`(**in-out** l; **in** `subContractor`, X, C, ϵ)

```
if ¬l.precise then
  if ¬l.certified then
    subContractor(l.box)
  end if
  if l.box ≠ ∅ and I-Newton(l.box,X) then
    l.certified ← true
  end if
  if maxDiameter(l.box) < ε then
    l.precise ← true
  end if
end if
```

La procédure `contract` est paramétrée principalement par la procédure de contraction `subContractor` utilisée (`HC4` ou `3BCID` dans nos expérimentations). Ce sous-contracteur porte sur le système C d'équations, c'est-à-dire sur le sous-système $k \times k$ (pour `Box-k-Revise`) ou sur le système entier (pour `S-k-B-Revise`). Après un appel à `subContractor`, `I-Newton` est appelé sur le sous-système $k \times k$. Si `I-Newton` certifie qu'il existe une solution unique dans une feuille, c.-à-d. si `I-Newton` contracte *l.box* et renvoie *true*, cette feuille est étiquetée comme *certified*.

4.2 Réutilisation de l'arbre local (procédure `UpdateLocalTree`)

Une version plus simple de l'algorithme 1 n'appelait pas la procédure `UpdateLocalTree` et initialisait simplement la liste L avec la boîte courante. Cependant, plutôt que de réaliser un effort de recherche intensif dans un seul sous-système, nous avons préféré propager rapidement les réductions obtenues aux autres sous-systèmes. Ainsi, la procédure `UpdateLocalTree` réutilise l'arbre local (c.-à-d., ses feuilles) qui a été sauvé lors d'un appel précédent à l'algorithme 1. Chaque feuille dans la liste courante L est alors mise à jour en l'intersectant avec la boîte courante et en la contractant avec `subContractor`.

Algorithm 3 `UpdateLocalTree` (**in-out** L; **in** `box`, X, C, ϵ, `subContractor`)

```
if L = ∅ then
  /* Initialise la racine de l'arbre local avec la boîte courante */
  L ← {Leaf(box)}
else
  for all l ∈ L do
    /* Met à jour et contracte chaque feuille de l'arbre local */
    if l.box ≠ (l.box ∩ box) then
      l.box ← l.box ∩ box
      contract(l, subContractor, C, ε))
      if l.box = ∅ then L.remove(l) end if
    end if
  end for
end if
```

En fait, les feuilles des arbres locaux sont également sauvegardées dans l'arbre de recherche global. A cette fin, la liste L est implantée comme une structure de données *réversible* (*"backtrackable"*) mise à jour en cas de retour arrière. Cela évite de refaire plusieurs fois le même travail dans les sous-systèmes, en particulier quand l'heuristique *multisplit* est utilisée (voir partie 5).

4.3 Traitement paresseux des feuilles

Les premières expérimentations nous ont montré que le traitement d'une feuille dans un arbre local,

c'est-à-dire le fait de la bissecter et de contracter les deux sous-boîtes, était souvent contre-productif. Nous avons alors défini un ratio entrée/sortie ρ_{io} pour décider si une feuille donnée devait être traitée dans l'arbre local.

$$\rho_{io}(B, I, O, F) = \frac{Max_{x \in I}(smear(x))}{Max_{x \in O}(smear(x))}$$

La fonction `ProcessLeaf` calcule ρ_{io} sur une feuille. Si ce ratio est plus grand qu'un seuil $\tau_{\rho_{io}}$, la feuille n'est alors pas traitée par la procédure *revise* courante.

ρ_{io} est basé sur la fonction *smear* [11] définie par : $smear(x) := Max_{f \in F}(|\frac{\partial f}{\partial x}| \times Diam(x))$. Cette fonction est souvent utilisée pour choisir la prochaine variable à bissecter dans les NCSP (celle qui a la plus grande évaluation pour cette fonction *smear*).

On peut alors expliquer le dénominateur de ρ_{io} de la manière suivante : les variables de sortie avec une grande évaluation *smear* (entraînant un petit ratio ρ_{io}) indiquent une grande contraction potentielle quand elles sont bissectées dans l'arbre local au sous-système. Désirer un petit impact des variables d'entrée est moins intuitif. Nous comprenons que des domaines d'entrée larges conduisent généralement à des domaines de sortie (correspondant aux boîtes de feuilles) larges aussi et donc à une faible réduction. Cet argument est aussi valable pour les dérivées des fonctions. Pour illustrer ce point, prenons un sous-système de taille 1 comme $0.001\,y + x^2 - 1 = 0$ (x est la variable de sortie ; $[x] = [y] = [-1, 1]$), avec $\rho_{io} = \frac{0.002}{4} = 0.0005$. Après une bissection sur x, la contraction de ce sous-système produit un tout petit intervalle pour x. On obtiendrait un grand intervalle pour x si l'on considérait le sous-système $y + x^2 - 1 = 0$ avec $\rho_{io} = \frac{2}{4} = 0.5$.

4.4 Paramètres utilisateurs de la procédure *revise*

Il est apparu dans les expérimentations que le réglage de $\tau_{\rho_{io}}$ avait un impact important sur le temps de calcul, de telle sorte que ce paramètre est devenu le principal paramètre à régler pour les algorithmes de Box-k ou S-kB consistance. Les deux autres paramètres à fixer sont `subContractor` et τ_{leaves}. Les expérimentations décrites en partie 6 montrent que le sous-contracteur `3BCID` est généralement un bon choix, et τ_{leaves} a été fixé empiriquement, son impact sur le temps de calcul étant beaucoup moins important que celui de $\tau_{\rho_{io}}$.

4.5 Propriétés de la procédure *revise*

La proposition suivante formalise la correction et les complexités en mémoire et en temps de la procédure `Boxk-SkB-Revise`.

Proposition 2 *Soit $P' = (X', C', B)$ un sous-système du CSP $P = (X, C, B)$, avec $|X| = n$, $|C| = m$, $|X'| = |C'| = k$.*

La procédure `Boxk-SkB-Revise`*, appelée avec $\tau_{leaves} = +\infty$ et $\tau_{\rho_{io}} = +\infty$, rend P' Box-k-consistant si l'ensemble des équations est C' (respectivement S-kB-consistant si l'ensemble des équations est C).*

Soit Diam le plus grand diamètre des intervalles de B. Soit $d = log_2(\frac{Diam}{\epsilon})$, le nombre maximum de fois qu'un intervalle donné doit être bissecté pour atteindre la précision requise ϵ [1].

La complexité en mémoire de `Boxk-SkB-Revise` *est $O(k\,\tau_{leaves})$.*

Le nombre d'appels à `subContractor` *est $O(k\,d\,\tau_{leaves})$.*

Preuve. La correction se base sur le processus combinatoire réalisé par la procédure `Boxk-SkB-Revise`. Appelée avec le sous-système de contraintes C' et avec $\tau_{leaves} = +\infty$, la procédure calcule toutes les boîtes atomiques de précision ϵ dans le sous-système avant de retourner leur enveloppe, réalisant ainsi la consistance globale de P'.

La complexité en mémoire vient de la recherche en largeur d'abord qui doit stocker les $O(\tau_{leaves})$ feuilles de l'arbre local. La procédure *revise* traite des boîtes n-dimensionnelles, mais pour économiser de la mémoire, elle ne stocke que les k intervalles d'un sous-système $k \times k$.

Le nombre d'appels au sous-contracteur est borné par le nombre de nœuds dans l'arbre local. Le nombre de feuilles de cet arbre est τ_{leaves} (correspondant aux boîtes *vivantes* qui peuvent contenir des solutions) plus le nombre de feuilles *mortes* éliminées par filtrage et qui se ramassent à la pelle. Pour chaque feuille vivante l, le nombre de nœuds créés dans l'arbre pour atteindre l est au plus $2 \times d \times k$ car la racine doit être au plus bissectée d fois dans chacune de ses k dimensions. Bien que de nombreux nœuds internes sont partagés par plusieurs feuilles vivantes, cela borne le nombre d'appels à l'opérateur de sous-filtrage par $O(k\,d\,\tau_{leaves})$. □

5 Découpage multidimensionnel

Il est apparu que les procédures `Box-k` et `S-kB` *revise* ont non seulement un effet de contraction, mais donnent la perspective d'une nouvelle manière de faire des points de choix, c'est-à-dire de construire l'arbre de recherche global. Cette nouvelle stratégie de découpage de l'espace de recherche est appelée découpage multidimensionnel (en abrégé *multisplit*).

[1] d est généralement compris entre 20 et 60 dans les NCSP existants.

Définition 8 *Considérons un sous-système $k \times k$ P' défini dans un NCSP $P = (X, C, B)$. Considérons un ensemble S de m boîtes associé à P' tel que S contienne toutes les solutions de P et que les m boîtes obtenues par projection sur P' des boîtes de S soient deux à deux disjointes.*

Un multisplit de dimension k consiste à découper l'espace de recherche B en les m boîtes de S.

En pratique, les m boîtes correspondent aux feuilles de l'arbre local d'un sous-système. A la fin d'une propagation `S-kB`, notre stratégie de résolution fait un choix entre une bissection classique et un *multisplit*. Si tous les sous-systèmes ont un ratio ρ_m supérieur à un seuil donné τ_m, alors on effectue une bissection standard. Sinon, nous choisissons le sous-système avec le plus petit ρ_m, et nous remplaçons la boîte courante par l'ensemble L des m feuilles de l'arbre local.

$$\rho_m = \frac{\sum_{l \in L} Volume(l)}{Volume(Hull(L))}$$

La procédure *multisplit* généralise une procédure utilisée par `IBB` (cf. partie 6.1). `IBB` réalise un *multisplit* quand il a trouvé les m solutions dans un bloc donné. La différence ici est qu'un *multisplit* peut avoir lieu avec des boîtes dont la taille n'a pas atteint la précision requise.

6 Expérimentations

Les algorithmes `Box-k` et `S-kB` ont été implantés dans l'outil de résolution de contraintes sur intervalles `Ibex`, bibliothèque logicielle libre écrite en `C++` [5, 4]. La variante avec *multisplit* (`msplit`) réalise un découpage multidimensionnel du sous-système avec le ratio ρ_m minimum, pourvu que $\rho_m < \tau_m$=0.99. Tous les compétiteurs sont implantés dans la même bibliothèque, ce qui rend la comparaison équitable.

6.1 Expérimentations sur des problèmes décomposés

Dix problèmes *décomposés*, décrits dans [17, 16], apparaissent dans le tableau 1. Ils ont été auparavant décomposés par des algorithmes de graphe (*eq*) comme le couplage maximal, ou par des algorithmes géométriques utilisant la rigidité (*geo*). Ils peuvent être résolus efficacement par la méthode `IBB` [17, 16].

Brève description de `IBB`

`IBB` est dédié à des systèmes décomposés, c'est-à-dire des systèmes peu denses d'équations qui ont préalablement été décomposés en une séquence de blocs/sous-systèmes bien contraints irréductibles [1]. L'algorithme de retour arrière inter blocs (`IBB`) traite chaque bloc dans l'ordre de la séquence. Il entrelace des étapes de contraction (réalisés par `HC4` et `I-Newton`) et des bissections à l'intérieur du bloc jusqu'à ce que des boîtes atomiques (solutions du bloc) soient obtenues. Des points de choix sont alors effectués : les variables du bloc sont remplacées par une des boîtes atomiques et sont donc considérées comme constantes dans les blocs suivants.

La procédure `Box-k-Revise` plus *multisplit* représente une généralisation de `IBB` dans le sens que les domaines des variables d'entrée d'un sous-système ne sont plus nécessairement atomiques et qu'un *multisplit* n'est pas toujours réalisé après le traitement d'un sous-système. En d'autres termes, le traitement des blocs par IBB n'est pas une procédure *revise*, c'est une procédure *ad hoc* dans un algorithme dédié aux systèmes décomposés. Appliquée aux systèmes décomposés, notre nouvelle approche n'exploite pas par ailleurs l'ordre entre les blocs qui procure à IBB une heuristique de découpage utile.

Protocole expérimental

Chaque stratégie `Box-k` a été réglée avec 6 jeux de valeurs différents pour les paramètres : $\tau_{\rho_{io}}$ est 0.01, 0.2 ou 0.8 (0.01 est toujours la meilleure valeur pour les systèmes décomposés ; la précision ρ_{propag} utilisée dans la propagation `HC4` est 1% ou 10% ; Tous les autres paramètres ont été fixés empiriquement : la précision ρ_{propag} dans la propagation `Box-k` est toujours 10% ; le nombre maximum τ_{leaves} de feuilles dans un arbre local est 10 ; le nombre de tranches de `3BCID` dans `S-kB(3BCID)` est 10. Pour être équitable, les paramètres des algorithmes compétiteurs ont été réglés de sorte que 8 essais ont été réalisés pour `Box` et `HC4`, et 16 essais pour `3BCID`. Pour tous les tests, le déclenchement de I-Newton (taille du diamètre maximum à partir duquel I-Newton est lancé) est 10, et le même ordre des variables est utilisé pour la bissection dans une stratégie à tour de rôle (sauf pour `IBB` et pour `Box-k` avec *multisplit*, qui ont leurs propres heuristiques).

Les sous-systèmes gérés par la propagation sont définis automatiquement. Les blocs irréductibles produits par la décomposition de `IBB` constituent les sous-systèmes gérés par la propagation `Box-k`.

Résultats

Les stratégies basées sur `HC4`, `Box` et `3BCID` suivies par `I-Newton` ne sont pas du tout compétitives avec `Box-k` et `IBB` pour les systèmes décomposés testés. La comparaison de `Box-k`[2] avec `IBB` est très positive

[2]En nous basant sur nos précédents travaux sur `IBB`, nous avons concentré l'effort à l'intérieur des sous-systèmes, écartant la procédure `S-kB` revise pour cette catégorie de problèmes.

TAB. 1 – Résultats expérimentaux sur les problèmes IBB. Les 3 premières colonnes indiquent le nom du système, son nombre de variables et son nombre de solutions. Les 3 colonnes suivantes donnent le temps de calcul (en haut) et le nombre de boîtes (en bas), c.-à-d., le nombre de points de choix, obtenus sur un processeur Intel 6600 2.4 GHz par les stratégies existantes basées sur HC4, Box ou 3BCID suivies par I-Newton (entre deux bissections réalisées avec une heuristique à tour de rôle pour le choix de variable). Les 4 dernières colonnes donnent les résultats obtenus par nos algorithmes sur la même machine : Box-k paramétré par subContractor=HC4 ou subContractor=3BCID, avec *multisplit* (msplit) ou sans. Seule une propagation Box-k est lancée entre deux bissections.

Problème	#vars	#sols	**HC4**	**Box**	**3BCID**	**IBB**	**BoxK(HC4)**		**BoxK(3BCID)**	
								msplit		**msplit**
Chair(eq)	178	8	>3600	>3600	>3600	**0.27**	>3600	16.5*	>3600	**0.52**
1x15,1x13,1x9,5x8,3x6,...								575		15
Latham(eq)	102	96	>3600	>3600	39.9	**0.17**	**0.94**	1.35	1.5	1.08
1x13,1x10,1x4,25x2,25x1					587		839	199	991	189
Ponts(eq)	30	128	33.4	33.4	1.89	**0.59**	6.85	8.19	0.79	**0.71**
1x14,6x2,4x1			20399	20399	357		783	231	307	231
Ponts(geo)	38	128	44.1	44.1	2.6	**0.16**	2.01	0.31	1.45	**0.39**
13x2,12x1			18363	18363	685		6711	767	6711	767
Sierp3(geo)	124	198	>3600	>3600	77.5	**0.62**	49.0	**1.38**	52.5	1.77
44x2,36x1					1727		84169	1513	84169	1513
Star(eq)	46	128	>3600	>3600	4.9	**0.05**	35.6	**0.12**	44.0	0.26
3x6,3x4,8x2					283		44195	263	44023	263
Tangent(eq)	28	128	77	77	2.1	**0.08**	1.74	**0.08**	1.87	0.14
1x4,10x2,4x1			390903	390903	753		12027	255	12235	255
Tangent(geo)	42	128	–	–	7.38	**0.08**	0.80	0.19	0.80	**0.19**
2x4,11x2,12x1					859		1415	251	1407	251
Tetra(eq)	30	256	1281	1281	12.3	**0.63**	33.6	1.06	13.57	**0.76**
1x9,4x3,1x2,7x1			607389	607389	1713		4619	483	2243	483
Sierp3(eq)	cf. tableau 2					>5000	cf. tableau 2			

car les temps de calcul pour IBB sont réellement les meilleurs temps obtenus par toutes les variantes de cet algorithme. De plus, aucun *timeout* n'est atteint par Box-k+multisplit et IBB est en moyenne seulement 2 fois plus rapide que Box-k (au plus 6 sur Latham). Comme prévu, les résultats confirment que le découpage multidimensionnel est toujours pertinent pour les problèmes décomposés.

Pour le problème Sierp3(eq) (la fractale de Sierpinski au niveau 3) une décomposition équationnelle fait apparaître un gros bloc irréductible 50×50 de contraintes de distance. Cela rend IBB avec HC4 inefficace sur ce problème. Il est cependant à noter qu'une variante de IBB avec un filtrage entre blocs (IBF) [16] utilisant 3B (variante confidentielle car souvent inefficace) réussit à résoudre ce problème en 330 secondes.

6.2 Expérimentations sur des systèmes structurés

Huit systèmes *structurés* apparaissent dans le tableau 2. Ce sont des chaînes de contraintes d'arité raisonnable [14]. Ils sont appelés *structurés* car ils ne sont pas suffisamment creux pour être décomposés , c'est-à-dire que le système contient un seul bloc irréductible, rendant IBB sans objet. Une brève analyse manuelle du graphe de contraintes pour chaque problème nous a conduits à définir quelques sous-systèmes bien contraints de taille raisonnable (entre 2 et 10). De la même façon, nous avons remplacé le bloc 50×50 de Sierp3(eq) par des sous-systèmes 6×6 et 2×2 pour S-kB.

Les stratégies standard basées sur HC4 ou Box suivies par I-Newton ne sont généralement pas compétitives avec S-kB sur les problèmes testés. La comparaison avec 3BCID est bonne, le gain variant entre 0.7 et 12. Notre stratégie de résolution basée sur S-kB apparaît donc comme étant un algorithme hybride robuste, jamais très loin derrière 3BCID et quelquefois nettement meilleur. Le nombre de boîtes obtenu souligne le pouvoir de filtrage additionnel apporté par nos algorithmes traitant les sous-systèmes bien contraints. De nouveau, autoriser les multisplits semble être la meilleure option.

6.3 Apports des dispositifs sophistiqués dans les procédures *revise*

Nous montrons finalement les tableaux 3 et 4 pour évaluer les apports individuels de deux points : le paramètre $\tau_{\rho_{io}}$ utilisé par la procédure ProcessLeaf et la liste réversible des feuilles permettant de réutiliser le travail effectué dans les sous-systèmes.

Pour chaque case dans les deux tableaux, on indique le meilleur résultat en temps obtenu avec les deux sous-contracteurs HC4 et 3BCID. La première ligne de résultats correspond à la procédure Boxk-SkB-Revise im-

TAB. 2 – Résultats sur les problèmes structurés. Le même protocole a été suivi, sauf qu'une propagation `S-kB` (et non `Box-k`) donne les meilleurs résultats quand elle suit une contraction `3BCID` et I-Newton (entre deux bissections).

Problème	#vars	#sols	**HC4**	**Box**	**3BCID**	**S-kB(HC4)**		**S-kB(3BCID)**	
							msplit		**msplit**
Bratu	60	2	58	626	48.7	47.0	**33.0**	135	126
29x3			15653	13707	79	39	17	43	25
Brent	10	1015	1383	127	17.0	28.5	20.2	44.9	31.0
2x5			7285095	42191	9849	2975	4444	4585	1309
BroydenBand	20	1	>3600	0.17	**0.11**	0.45	0.15	0.91	0.31
1x6,3x5				1	21	4	19	17	3
BroydenTri	30	2	1765	**0.16**	0.25	0.22	0.24	0.39	0.29
6x5			42860473	63	25	11	19	9	3
Reactors	30	120	>3600	>3600	288	340	315	81.4	**67.5**
3x10					39253	14576	10247	1038	788
Reactors2	10	24	>3600	>3600	28.8	**9.5**	12.3	10.4	12.2
2x5					128359	4908	10850	4344	5802
Sierp3(eq)	83	6	>3600	>3600	4917	>3600	>3600	>3600	**389**
1x14,6x6,15x2,3x1					44803				218
Trigexp1	30	1	>3600	13	**0.08**	**0.08**	**0.08**	**0.08**	0.09
6x5				27	1	1	1	1	1
Trigexp2	11	0	1554	>3600	83.7	**81.2**	85.7	105	83.0
2x4,2x3			2116259		16687	15771	16755	3797	2379

TAB. 3 – Apport sur les problèmes `IBB` de la structure de données réversible (BT) et de $\tau_{\rho_{io}}$ dans la stratégie `Box-k`. $\tau_{\rho_{io}} = \infty$ signifie que les feuilles du sous-système seront toujours traitées dans la procédure *revise*.

	Chair	**Latham**	**Ponts(eq)**	**Ponts(geo)**	**Sierp3(geo)**	**Star**	**Tan(eq)**	**Tan(geo)**	**Tetra**
BT, $\tau_{\rho_{io}}$	0.52	1.08	0.71	0.31	1.38	0.12	0.08	0.19	0.76
¬ BT, $\tau_{\rho_{io}}$	10.8	4.61	1.51	1.27	23.9	2.34	0.71	1.58	2.13
BT, $\tau_{\rho_{io}} = \infty$	23.4	4.71	2.60	1.00	23.8	1.67	1.09	1.81	3.57
¬ BT, $\tau_{\rho_{io}} = \infty$	24.2	6.60	2.80	1.11	23.9	2.40	1.15	1.82	3.54

plantée ; les lignes suivantes correspondent à des versions plus simples pour lesquelles au moins un des deux dispositifs a été enlevé ou les deux (dernière ligne). Le multisplit est permis dans tous les tests.

On peut faire trois observations principales. D'abord, quand on observe un gain significatif apporté par ces dispositifs sur un système donné, alors ce système est traité efficacement par rapport aux compétiteurs (cf. tableaux 1 et 2). Ensuite, $\tau_{\rho_{io}}$ semble avoir un plus grand impact sur la performance que la liste réversible mais la différence est faible.

Enfin, plusieurs systèmes sont seulement légèrement améliorés par un des deux points tandis que le gain est significatif quand les deux sont mis ensemble. Ceci est vrai pour la plupart des problèmes `IBB`. Sur ces systèmes, entre 2 bissections dans l'arbre de recherche, il arrive souvent que le travail à l'intérieur de plusieurs sous-systèmes conduise à identifier des boîtes atomiques (d'autres sous-systèmes ne sont pas complètement explorés grâce à $\tau_{\rho_{io}}$). Bien qu'un multisplit ne soit effectué que sur un seul des sous-systèmes, le travail réalisé sur les autres est sauvegardé grâce à la liste réversible.

7 Conclusion

Nous avons proposé de nouveaux algorithmes de filtrage pour traiter des sous-systèmes bien contraints $k \times k$ dans un NCSP. Des appels à un `I-Newton` $k \times k$ et des bissections à l'intérieur de tels sous-systèmes sont utiles pour mieux contracter les NCSP décomposés et structurés. De plus, les arbres de recherche locaux, à l'intérieur des sous-systèmes, permettent à la stratégie de recherche globale d'apprendre des points de choix intéressants consistant à couper plusieurs domaines simultanément.

Les stratégies de recherche basées sur les propagations `S-kB` ou `Box-k` et le découpage multi-dimensionnel ont trois paramètres principaux : le choix entre `Box-k` et `S-kB` (`Box-k` semble meilleur uniquement pour les systèmes décomposés), le choix du sous-contracteur (`3BCID` semble être souvent un bon choix), et $\tau\rho_{io}$. Ce dernier paramètre apparaît comme le plus important à régler.

Les premières expérimentations sur des systèmes décomposés et structurés ont montré que nos nouvelles stratégies de résolution étaient plus efficaces que les stratégies standard basées sur `HC4`, `Box` ou `3BCID` (avec `I-Newton`) pour ces problèmes. `Box-k`+multisplit peut

TAB. 4 – Apport sur les problèmes structurés de la structure de données réversible (BT) et de $\tau_{\rho_{io}}$ (dans la stratégie `S-kB`).

		Bratu	**Brent**	**BroyB.**	**BroyT.**	**Reac.**	**Reac.2**	**Sierp3B(eq)**	**Trigexp1**	**Trigexp2**
BT,	$\tau_{\rho_{io}}$	33.0	20.2	0.15	0.24	67.5	12.2	389	0.08	83.0
¬ BT,	$\tau_{\rho_{io}}$	33.2	21.0	0.14	0.23	97.7	12.0	411	0.07	85.0
BT,	$\tau_{\rho_{io}} = \infty$	33.9	23.8	0.38	0.28	164	13.1	519	0.1	103
¬ BT,	$\tau_{\rho_{io}} = \infty$	33	28.7	0.40	0.38	401	18.7	533	0.07	148

être vu comme une généralisation de `IBB`. Il peut aussi résoudre des systèmes décomposés de grande taille avec des blocs assez petits en moins d'une seconde, et peut par ailleurs traiter des systèmes structurés que `IBB` ne peut pas résoudre.

Les sous-systèmes ont été ajoutés automatiquement pour les NCSP décomposés, mais manuellement pour les NCSP structurés. Dans cet article, nous avons validé le fait que le traitement de sous-systèmes pouvait apporter une contraction supplémentaire et des points de choix multi-dimensionnels pertinents. La prochaine étape est de sélectionner automatiquement un ensemble de sous-systèmes. Nous pensons que des algorithmes de graphe prenant en compte un critère similaire à ρ_{io} peuvent être adaptés pour conduire à des heuristiques efficaces.

Références

[1] S. Ait-Aoudia, R. Jegou, and D. Michelucci. Reduction of Constraint Systems. In *Compugraphic*, 1993.

[2] F Benhamou, Goualard F., L. Granvilliers, and J.-F. Puget. Revising Hull and Box Consistency. In *Proc. of ICLP*, pages 230–244, 1999.

[3] C. Bliek, B. Neveu, and G. Trombettoni. Using Graph Decomp. for Solving Continuous CSPs. In *Proc. CP, LNCS 1520*, pages 102–116, 1998.

[4] G. Chabert. `www.ibex-lib.org`, 2009.

[5] G. Chabert and L. Jaulin. Contractor Programming. *Artificial Intelligence*, Available online 18 March 2009.

[6] H. Collavizza, F. Delobel, and M. Rueher. Comparing Partial Consistencies. *Reliable Computing*, 5(3) :213–228, 1999.

[7] J. Cruz and P. Barahona. Global Hull Consistency with Local Search for Continuous Constraint Solving. In *Proc. EPIA, LNAI 2258*, pages 349–362, 2001.

[8] R. Debruyne and C. Bessière. Some Practicable Filtering Techniques for the Constraint Satisfaction Problem. In *Proc. IJCAI*, pages 412–417, 1997.

[9] A.L. Dulmage and N.S. Mendelsohn. Covering of Bipartite Graphs. *Canadian Journal of Mathematics*, 10 :517–534, 1958.

[10] L. Jaulin, M. Kieffer, O. Didrit, and E. Walter. *Applied Interval Analysis*. Springer, 2001.

[11] R.B. Kearfott and M. Novoa III. INTBIS, a portable interval Newton/Bisection package. *ACM Trans. on Mathematical Software*, 16(2) :152–157, 1990.

[12] Y. Lebbah, C. Michel, M. Rueher, D. Daney, and J.P. Merlet. Efficient and safe global constraints for handling numerical constraint systems. *SIAM Journal on Numerical Analysis*, 42(5) :2076–2097, 2005.

[13] O. Lhomme. Consistency Tech. for Numeric CSPs. In *IJCAI*, pages 232–238, 1993.

[14] J-P. Merlet. `www-sop.inria.fr/coprin/ logiciels/ALIAS/Benches/benches.html`, 2009.

[15] A. Neumaier. *Interval Methods for Systems of Equations*. Cambridge Univ. Press, 1990.

[16] B. Neveu, G. Chabert, and G. Trombettoni. When Interval Analysis helps Interblock Backtracking. In *Proc. CP, LNCS 4204*, pages 390–405, 2006.

[17] B. Neveu, C. Jermann, and G. Trombettoni. Inter-Block Backtracking : Exploiting the Structure in Continuous CSPs. In *Selected papers WS COCOS, LNCS 3478*, pages 15–30, 2005.

[18] J.-C. Régin. A Filtering Algorithm for Constraints of Difference in CSPs. In *Proc. AAAI'94*, pages 362–367, 1994.

Un algorithme de décision dans l'algèbre des arbres finis ou infinis et des queues

Thi-Bich-Hanh Dao

Laboratoire d'Informatique Fondamentale d'Orléans – Université d'Orléans
Bâtiment 3IA, Rue Léonard de Vinci – 45067 Orléans Cedex 2
thi-bich-hanh.dao@univ-orleans.fr

Résumé

Les structures des arbres, des listes, des piles et des queues sont souvent utilisées en programmation logique et programmation logique avec contraintes. Il est donc important de pourvoir résoudre des contraintes dans ces structures. Les listes et les piles peuvent être considérées comme des cas spéciaux des arbres, mais ce n'est pas le cas des queues avec les opérations d'ajout d'un élément à gauche ou à droite. Nous présentons dans ce papier un algorithme de décision dans l'algèbre des arbres finis ou infinis étendus avec des queues. De cet algorithme découle un résultat en logique : la complétude et la décidabilité de la théorie du premier ordre de cet algèbre.

Abstract

The structures of trees, lists, stacks and queues are usually used in logic programming or constraint logic programming. It is then important to be capable to solve constraints in these structures. Lists and stacks can be considered as special cases of trees, but it is not the case for queues with left- and right-insert operations. We present in this paper a decision algorithm in the algebra of finite or infinite trees extended with queues. This algorithm gives a result in logic : the completeness and the decidability of the first-order theory of this algebra.

1 Introduction

Dans la programmation logique (PL) ou la programmation logique avec contraintes (PLC), la structure d'arbres et les structures de listes, de piles ou de queues sont souvent utilisées. On se trouve donc face à décider ou à résoudre des contraintes dans ces structures. Les listes et les piles avec l'opération d'ajout d'un élément à gauche pour les listes et à droite pour les piles peuvent être considérées comme des cas spéciaux des arbres. Une procédure de décision d'une algèbre des arbres s'applique donc habituellement aux arbres avec listes et piles. Les queues sont des structures où nous pouvons ajouter un élément à la fin ou bien prendre l'élément du début. Ces deux opérations correspondent respectivement à des ajouts à droite et à gauche d'un élément dans une queue. Les queues avec ces opérations ne peuvent pas être considérées comme des arbres, car une queue peut être construite de différentes façons, ce qui contrarie une des propriétés principales des arbres. Par exemple une liste $[a, b]$ est construite par l'unique construction $[a|[b|[]]]$, mais une queue $\langle a, b\rangle$ a plusieurs constructions différentes, par exemple en ajoutant a à gauche de la queue vide puis b à droite ou en ajoutant b à gauche de la queue vide puis a à gauche. Une procédure de décision dans les arbres ne peut donc pas s'appliquer aux queues. Tandis que la décidabilité de l'algèbre des arbres finis avec des queues est connu [13], celle des arbres finis ou infinis avec des queues à notre connaissance, reste une problème ouvert.

Un simple exemple d'utilisation de queues est dans l'analyse syntaxique, où l'on veut reconnaître les mots $a^n b^n$ générés par les règles $S \to \varepsilon \mid aSb$. Dans des implantations de la PL ou la PLC, avec des grammaires de clauses définies, les queues sont souvent représentées par des différences de listes. Une différence de liste est un terme $L1 - L2$, qui signifie la liste obtenue en coupant la liste $L2$ de la fin de $L1$, par exemple $[1, 2, 3, 4] - [3, 4]$ représente la liste $[1, 2]$. Voici un programme Prolog utilisant des différences de listes :

```
s(X0-X0).
s([a|X0]-X1) :- s(X0-[b|X1]).
```

Cependant, la différence de listes n'est qu'une astuce pour gérer la fin d'une liste. Nous nous plaçons ensuite dans l'algèbre des arbres étendus avec des queues. L'ensemble des symboles de fonctions est étendu de la

constante *nil* pour la queue vide et des symboles *al* et *ar* correspondant respectivement à l'ajout à gauche et l'ajout à droite d'un élément dans une queue. Les clauses seront plus simples et naturelles :

```
s(nil).
s(al(a,ar(X,b))) :- s(X).
```

Les arbres avec un ensemble infini de symboles de fonction modélisent la base de la PL ou de la PLC [2, 7]. L'algèbre qui modélise cette extension sera celle des arbres finis ou infinis étendus avec des queues, définis sur un ensemble infini de symboles de fonction. Il est important donc d'assurer la décidabilité dans cet algèbre. La décidabilité de cette extension a également un intérêt dans autres domaines de l'informatique comme la logique ou la vérification.

La décision dans des algèbres des arbres font déjà l'objet de plusieurs études. Dans le cas des arbres finis nous référons aux travaux de A. Malcev [11], K. L. Clark [1], K. Kunen [8] et H. Comon [3]. Pour les arbres finis ou infinis, M. Maher a proposé une axiomatisation complète ainsi qu'une procédure de décision [10]. Un algorithme de résolution de contraintes du premier ordre dans les algèbres des arbres est proposé en [4, 6] et qui fait d'office la décision pour les formules closes.

Des extensions d'arbres avec d'autres structures sont étudiées dans différents travaux. Dans le cas d'extension de la théorie des arbres avec une théorie dite flexible, par exemple la théorie des rationnels additifs, une axiomatisation ainsi qu'une procédure de décision sont proposées [5]. Cependant, les arbres avec des queues n'entrent pas dans ce cadre. De plus, l'extension avec des queues n'est pas simplement d'avoir des queues en plus des arbres, mais une structure où un arbre peut avoir une partie qui est une queue, ou les éléments d'une queue peuvent être des arbres ou de nouveau des queues. Par conséquent les méthodes générales de décision dans une combinaison de théories comme Nelson-Oppen [12] ne peuvent non plus s'appliquer. Dans le cadre de la vérification, T. Rybina et A. Voronkov ont proposé un algorithme de décision dans l'algèbre des arbres finis avec queues [13]. Cependant l'algèbre étudiée est des arbres finis construits sur un ensemble fini de symboles de fonction, ce qui fait que des techniques utilisées pour la décision ne peuvent pas s'appliquer dans notre cas.

La contribution de ce papier est un algorithme de décision des formules du premier ordre dans cette algèbre. L'algorithme permet à conclure la décidabilité de la théorie du premier ordre de cette algèbre.

Le reste du papier est organisé comme suit. L'algèbre des arbres avec queues ainsi que ses propriétés sont présentées dans la section 2. Dans la section 3, nous présentons les formes résolue et basique et l'algorithme de transformation en forme résolue. Dans la section 4, nous présentons l'algorithme de décision dans l'algèbre et déduisons la décidabilité de la théorie du premier ordre de cette algèbre. Plusieurs exemples sont donnés pour illustrer l'algorithme. La section 5 est réservée à la conclusion. A cause des contraintes d'espace, les preuves ne sont pas présentes. Une version complète est accessible en ligne à la page des rapports de recherche du Laboratoire d'Informatique Fondamentale d'Orléans (http ://www.univ-orleans.fr/lifo/).

2 L'algèbre des arbres avec queues

Langage Soit S un ensemble de deux *sortes arbre* et *queue*. Soit V un ensemble de variables dont chacune est associée à une sorte fixée. On suppose que chaque sorte a un nombre infini de variables. On appelle un *type* chaque expression de la forme $\alpha_1 \times \cdots \times \alpha_n \rightarrow \beta$, où $\alpha_1, \ldots, \alpha_n$ et β sont des sortes de S. Soit F un ensemble infini de symboles de fonction. A chaque élément de F est associé un type $\alpha_1 \times \cdots \times \alpha_n \rightarrow arbre$ et le nombre n est appelé l'*arité* de f. On se donne trois autres symboles al, ar et ε où al est de type $\alpha \times queue \rightarrow queue$, ar est de type $queue \times \alpha \rightarrow queue$ et ε de type $\rightarrow queue$, avec $\alpha \in S$. Soit R un ensemble de symboles de relation contenant deux symboles *queue* et *arbre* d'arité 1.

Un *terme* de sorte β est soit une variable de cette sorte, soit de la forme $f(u_1, ..., u_n)$ où $f \in F$ de type $\alpha_1 \times \cdots \times \alpha_n \rightarrow \beta$ et $u_1, \ldots, u_n$ des termes de sortes $\alpha_1, \ldots, \alpha_n$ respectivement. Une *formule* est une expression de l'une des formes suivantes :

$$s = t,\ arbre(x),\ queue(t),\ true,\ false,$$
$$\neg\varphi, (\varphi \wedge \psi), (\varphi \vee \psi), (\varphi \rightarrow \psi), (\varphi \leftrightarrow \psi), \exists x\varphi, \forall x\varphi.$$

Ici t, s sont des termes et φ, ψ sont des formules de taille plus petite. Nous utiliserons la notation $\overline{x}$ pour désigner un vecteur de variable $x_1...x_n$, $\exists\overline{x}$ pour $\exists x_1...\exists x_n$ et $\forall\overline{x}$ pour $\forall x_1..\forall x_n$. Nous utiliserons la notation $[u_1..u_n]t$ ou $[\overline{u}]t$ pour désigner le terme $al(u_1, ..., al(u_n, t)....)$ et la notation $t[u_1..u_n]$ ou $t[\overline{u}]$ pour le terme $ar(...ar(t, u_1)..., u_n)$. Nous utiliserons la notation $\exists?$ et $\exists!$ pour exprimer "'il existe au plus une valeur" et "il existe exactement un", respectivement.

Structure $\mathcal{S}$ Nous définissons la structure $\mathcal{S}$, appelée l'*algèbre des arbres finis ou infinis avec queues* comme suit. Le domaine de $\mathcal{S}$ est l'ensemble $D = A \cup Q$, où A est l'ensemble des arbres, qui sont finis ou infinis, dont les nœuds sont étiquetés par les éléments de F, et Q l'ensemble des queues d'éléments de $A \cup Q$. Une queue est une suite d'éléments, à laquelle nous pouvons ajouter un élément à gauche (au début) ou à droite (à la fin). Les arbres et les queues représentent deux sortes

arbre et *queue* respectivement. Nous définissons maintenant l'interprétation de chaque symbole de fonction.

Pour chaque symbole de fonction f de type $\alpha_1 \times \cdots \times \alpha_n \rightarrow$ *arbre*, l'interprétation $\underline{f}$ de f est l'opération de construction d'arbre défini comme suit : avec $a_1, \ldots, a_n$ des éléments de sortes $\alpha_1, \ldots, \alpha_n$, $\underline{f}(a_1, \ldots, a_n)$ est un arbre dont la racine est étiquetée par f et les fils immédiats sont $a_1, \ldots, a_n$. Le symbole ε est interprété par la queue vide. Le symbole al est interprété par la fonction d'ajout à gauche $\underline{al} : D \times Q \rightarrow Q$ telle qu'avec $a \in D$, $\langle b_1, .., b_n \rangle \in Q$, $\underline{al}(a, \langle b_1, .., b_n \rangle) = \langle a, b_1, .., b_n \rangle$. Le symbole ar est interprété par la fonction d'ajout à droite $\underline{ar} : Q \times D \rightarrow Q$ telle qu'avec $\langle b_1, .., b_n \rangle \in Q$, $a \in D$, $\underline{ar}(\langle b_1, .., b_n \rangle, a) = \langle b_1, .., b_n, a \rangle$.

Propriétés des arbres Elles sont formulées par l'ensemble des schémas d'axiomes suivants [10, 6] :

$$\begin{array}{ll} \forall \overline{x} \forall \overline{y}\ \neg(f(\overline{x}) = g(\overline{y})) & (a1) \\ \forall \overline{x} \forall \overline{y}\ (f(\overline{x}) = f(\overline{y}) \rightarrow \bigwedge_i x_i = y_i) & (a2) \\ \forall \overline{x} \exists! \overline{z}\ \bigwedge_i z_i = t_i(\overline{x}\overline{z}) & (a3) \end{array}$$

Ici $f, g \in F$ et sont distincts, $\overline{x}$ un vecteur de variables x_i, $\overline{y}$ un vecteur de variable y_i, $\overline{z}$ un vecteur de variables distinctes z_i de sorte *arbre* et $t_i(\overline{x}\overline{z})$ un terme de la forme d'un symbole de fonction suivi de variables prises dans $\overline{x}$ et $\overline{z}$. La forme $(a1)$ est appelée *conflit de symboles* qui indique que les arbres construits par des symboles de fonction différents sont différents. La forme $(a2)$ est appelée *explosion* qui indique que les arbres égaux doivent avoir l'égalité entre les sous-arbres respectifs. La forme $(a3)$ est appelée *solution unique*, qui indique que certaines formes de conjonctions d'équations ont une solution unique. Par exemple la formule $x = f(x)$ a une solution unique pour x, qui est l'arbre infini $f(f(f(...)))$.

Propriétés des queues Les queues satisfont les propriétés suivantes [13] :

$$\begin{array}{ll} \forall x \forall \overline{u} \neg(x = t(x\overline{u})) & (q1) \\ \forall x \forall u (\neg(\varepsilon = [u]x) \wedge \neg(\varepsilon = x[u])) & (q2) \\ \forall u_1 .. \forall u_n \forall v_1 .. \forall v_m \forall x \neg([u_1..u_n]x = x[v_1..v_m]) & (q3) \\ \forall x (x = \varepsilon \vee \exists y \exists u (x = [u]y) \vee \exists y \exists u (x = y[u])) & (q4) \\ \forall x \forall y \forall u \forall v ([u]x = [v]y \rightarrow u = v \wedge x = y) & (q5) \\ \forall x \forall y \forall u \forall v (x[u] = y[v] \rightarrow u = v \wedge x = y) & (q6) \\ \forall x \forall u \forall v (al(u, ar(v, x)) = ar(al(u, x), v)) & (q7) \\ \forall u_1 ... \forall u_n ([u_1..u_n]\varepsilon = \varepsilon[u_1..u_n]) & (q8) \end{array}$$

Ici $n \neq m$, x, y sont des variables de sorte *queue*, $t(x\overline{u})$ est un terme de sorte *queue* qui contient des occurrences de x. La propriété $(q1)$ exprime le fait que les queues sont de longueur finie et une queue ne peut se servir à définir elle-même, $q(2)$ et $(q3)$ le fait que les queues de longueurs différentes sont différentes, et de $(q4)$ à $(q8)$ la relation entre les ajouts à gauche et à droite dans une queue. Il est important de noter que $(q7)$ et $(q8)$ expriment le fait qu'une queue peut être construite de différentes façons, par exemple $\langle a, b \rangle$ est la queue construite par $al(a, ar(\varepsilon, b))$ ou par $ar(al(a, \varepsilon), b)$. Ces propriétés distinguent les queues des arbres.

De plus, les queues satisfont les propriétés suivantes concernant des mots. Les queues peuvent être considérées comme des mots finis sur D et les opérations d'ajout à gauche et à droite peuvent être considérées comme des concaténations d'un mot avec une lettre. Soit D^* l'ensemble des mots finis, D^+ l'ensemble des mots finis non vides sur D et ε le mot vide. Pour deux mots u et v, uv signifie le résultat de la concaténation du mot u avec le mot v. Pour un mot u, $|u|$ désigne sa longueur et u^n le résultat de la concaténation de n fois u, u^* (u^+) désigne l'ensemble $\{u^n \mid n \geq 0\}$ $(\{u^n \mid n > 0\})$. Un mot u est primitif si $u \neq v^n$ avec $n > 1$ pour tout mot v. Deux mots u et v sont conjugués s'il existe deux mots s, t avec $s \neq \varepsilon$ tels que $u = ts$ et $v = st$. Par exemple sur l'alphabet $\{a, b\}$, deux mots *baaba* et *ababa* sont conjugués.

Propriété 1 [9] *Soient $t, s \in D^+$ et x une variable. L'équation $tx = xs$ a des solutions si et seulement si t et s sont conjugués, c'est-à-dire qu'il existe un couple $(u, v) \in D^* \times D^+$ tel que $t = uv$ et $s = vu$. De plus, si t et s sont primitifs alors $(uv)^*u$ est l'ensemble des solutions de l'équation $tx = xs$.*

Propriété 2 *Soit $t = uv$ et $t' = v'u'$ deux mots primitifs dans D^+ avec $|t| \neq |t'|$. On a*

$$uvx = xvu \wedge u'v'x = xv'u' \rightarrow |x| < |t| + |t'|.$$

Propriété 3 *Soit $t = uv$ et $t' = u'v'$ deux mots primitifs dans D^+ avec $|t| = |t'|$. Si $|u| = |v|$ alors*

$$\begin{array}{ll} & uvx = xvu \wedge u'v'x = xv'u' \\ \rightarrow & (u = u' \wedge x = u) \vee (u = u' \wedge v = v' \wedge uvx = xvu) \end{array}$$

sinon $uvx = xvu \wedge u'v'x = xv'u' \rightarrow$ false.

Ces propriétés montrent qu'avec des équations dans des queues, nous pouvons exprimer des queues avec des motifs répétés. Par exemple d'après la propriété 1, l'équation $[ab]x = x[ba]$ correspond à un ensemble infini des queues finis $\langle a \rangle$, $\langle a, b, a \rangle$, $\langle a, b, a, b, a \rangle$, ... (la forme $(ab)^*a$), d'après la propriété 2, la conjonction $[ab]x = x[ba] \wedge [aba]x = x[aba]$ donne une solution unique $\langle a, b, a \rangle$, et d'après la propriété 3, la conjonction $[aba]x = x[aba] \wedge [aba]x = x[aab]$ n'a pas de solution.

3 Formules basiques

A partir d'ici nous supposons que les variables soient numérotées par des entiers naturels distincts, et pour une variable x soit $no(x)$ son numéro. Nous supposons que dans chaque formule et sous-formule, si x est une variable libre et y une variable quantifiée, alors $no(y) > no(x)$. Cette condition est satisfaisable car les variables quantifiées peuvent être renommées et l'ensemble de variables est infini.

En utilisant la propriété ($q7$) des queues, les termes construits avec les symboles al, ar sont réécrits dans la forme où les symboles les plus internes sont ar et ceux des plus externes sont al. Par exemple le terme $ar(ar(al(u_1, x), u_2), u_3)$ devient $al(u_1, ar(ar(x, u_2), u_3))$. La notation $[t_1..t_n]x[s_1..s_m]$ désignera ces termes et la variable x sera appelée le *noyau* du terme. Soit $\overline{u} = u_1..u_n$ et $1 \leq k \leq n$, on note par $\rho(k, \overline{u})$ la rotation circulaire à droite de k places de $\overline{u}$, c'est-à-dire $u_{k+1}..u_n u_1..u_k$. Notons que $\overline{u}$ et $\rho(k, \overline{u})$ sont conjugués.

Une *contrainte élémentaire* est
- soit une contrainte de sorte de la forme $queue(x)$ ou $arbre(x)$, avec x une variable,
- soit une équation de la forme $x = y$, avec x, y des variables,
- soit une équation de la forme $x = f(u_1, .., u_n)$ avec x et les u_i des variables et f de type $\alpha_1 \times \cdots \times \alpha_n \rightarrow arbre$,
- soit une équation de l'une des formes $x = t$ et $t = s$, où x est une variable, t et s sont des termes de l'une des formes ε, $[\overline{u}]\varepsilon[\overline{v}]$ et $[\overline{u}]y[\overline{v}]$, avec y une variable et $\overline{u}, \overline{v}$ des vecteurs de variables.

Une équation triviale est une équation de la forme $t = t$. Pour simplifier l'écriture, les équations de la forme $x[\overline{u}] = [\overline{v}]y$ sont réorganisées en $[\overline{v}]y = x[\overline{u}]$. Pour les équations de la forme $x = t$ ou $[\overline{u}]x = x[\overline{v}]$, x est appelée le *représentant* de l'équation. Une conjonction c de contraintes élémentaires est *complètement typée* si pour toute variable x ayant une occurrence dans c, soit $queue(x)$ soit $arbre(x)$ est dans c.

Soit c une conjonction de contraintes élémentaires et soit u une variable. On définit $Acc_c(u)$ l'ensemble des variables accessibles depuis u dans c comme suit :
- si c contient une équation non triviale de la forme $u = t$ ou $x = t$ avec $x \in Acc_c(u)$, alors les variables dans t sont dans $Acc_c(u)$,
- si c contient une équation non triviale de la forme $[\overline{u}]u = u[\overline{v}]$ ou $[\overline{u}]x = x[\overline{v}]$ avec $x \in Acc_c(u)$, alors les variables de $\overline{u}$ et $\overline{v}$ sont dans $Acc_c(u)$.

Définition 1 *Une conjonction c de contraintes élémentaires est sous* forme résolue *si et seulement si*

1. *chaque équation est de l'une des formes $x = t$, $[\overline{u}]x = x[\rho(k, \overline{u})]$, où x est une variable et $k \leq |\overline{u}|$,*
2. *les représentants des équations sont tous distincts,*
3. *pour chaque équation de la forme $x = y$, $no(x) > no(y)$,*
4. *c est complètement typée et les contraintes de sorte respectent le type des symboles de fonction dans c,*
5. *pour toute variable x de sorte queue ayant une occurrence dans c, $x \notin Acc_c(x)$.*

Dans l'algorithme de décision présenté dans la section 4, nous allons maintenir les formules sous la forme d'une disjonction de formules basiques, dont la définition est comme suit.

Définition 2 *Une* formule basique *est de la forme $\exists\overline{u}(c \wedge \bigwedge_{i \in I} \neg\exists\overline{u_i}c_i)$, avec I éventuellement vide, où*

1. *c et les c_i sont des conjonctions résolues,*
2. *pour chaque variable u de $\overline{u}$, il existe une variable x libre dans $\exists\overline{u}c$ telle que $u \in Acc_c(x)$,*
3. *pour chaque variable u de $\overline{u_i}$, il existe une variable x libre dans $\exists\overline{u_i}c_i$ telle que $u \in Acc_{c_i}(x)$.*

Remarque : puisque chaque variable quantifiée dans une formule basique doit être accessible depuis une variable libre, la seule formule basique sans variable libre est la formule *true*.

Notons que dans une formule basique, des conjonctions de contraintes élémentaires sont sous forme résolue. Nous présentons dans la sous-section 3.1 suivante l'algorithme permettant de transformer une conjonction de contraintes élémentaires en forme résolue.

3.1 Algorithme de transformation en forme résolue

Nous utiliserons la notation d'équation $[\overline{u}]x \stackrel{p}{=} x[\overline{v}]$ pour marquer que les mots $\overline{u}$ et $\overline{v}$ sont primitifs et la notation $[\overline{u}]x \stackrel{*}{=} x[\overline{v}]$ pour marquer que les mots $\overline{u}$ et $\overline{v}$ sont conjugués et primitifs.

L'algorithme est une combinaison de deux phases. La première phase transforme une conjonction de contraintes élémentaires complètement typée de sorte que les représentants des équations $x = t$ sont tous distincts. A la fin de cette phase, les représentants des équations de la forme $[\overline{u}]x = x[\overline{v}]$ ne sont pas encore distincts. La seconde phase traite ces équations, elle créera une disjonction de formules de la forme $\exists\overline{u}c$, où éventuellement sur c il sera nécessaire de relancer la phase 1 et la phase 2. Nous présentons dans ce qui suit les deux phases et montrons que bien que les deux phases puissent se lancer mutuellement, l'applications se termine toujours au bout d'un temps fini.

Première phase

- Concordance de type : assurer que le type des symboles de fonction est respecté, et qu'il n'y a pas de conflit de type.
- Equations de sorte *arbre* : Dans ces règles, a est une contrainte élémentaire, x, y sont des variables de sorte *arbre* avec $no(x) > no(y)$, les u_i, v_j des variables, t un terme de sorte *arbre*, f et g deux symboles de fonction différents.

$$\begin{aligned} false \wedge a &\implies false \\ x = x &\implies true \\ x = y &\implies y = x \\ x = f(u_1,..,u_n) \wedge x = g(v_1,..,v_m) &\implies false \\ x = f(u_1,..,u_n) \wedge x = f(v_1,..,v_n) &\implies \\ x = f(u_1,..,u_n) \wedge u_1 = v_1 \wedge \cdots \wedge u_n = v_n & \\ x = y \wedge x = t &\implies \\ x = y \wedge y = t & \end{aligned}$$

- Equations de sorte *queue* : dès qu'il existe une variable x de sorte *queue* dans c telle que $x \in Acc_c(x)$, la conjonction est évaluée à *false*.

$$\begin{array}{lrcl} 1 & false \wedge a & \implies & false \\ 2 & t = t & \implies & true \\ 3 & t = x & \implies & x = t \\ 4 & y = x & \implies & x = y \\ 5 & x = y \wedge e(x) & \implies & x = y \wedge e(y) \\ 6 & x = t \wedge x = s & \implies & x = t \wedge t = s \\ 7 & x = t \wedge e(x) & \implies & x = t \wedge e(t) \\ 8 & [u]t = [v]s & \implies & u = v \wedge t = s \\ 9 & t[u] = s[v] & \implies & u = v \wedge t = s \\ 10 & [\overline{u}]\varepsilon = r[\overline{v}] & \implies & \varepsilon[\overline{u}] = r[\overline{v}] \\ 11 & [\overline{u}]x = \varepsilon[\overline{v}] & \implies & [\overline{u}]x = [\overline{v}]\varepsilon \\ 12 & \varepsilon = [\overline{u}]x[\overline{v}] & \implies & false \\ 13 & [\overline{u}]x[\overline{v}] = \varepsilon & \implies & false \\ 14 & [\overline{u}]x = x[\overline{v}] & \implies & false \end{array}$$

Ici a est une contrainte élémentaire, t, s sont des termes de sorte *queue*, x, y des variables de sorte *queue*, u, v des variables et $\overline{u}, \overline{v}$ des vecteurs de variables. Dans les règles 3, 6 et 7, t, s sont des termes qui ne sont pas une variable. Dans les règles 4 et 5, $no(x) > no(y)$. Dans la règle 5, $e(x)$ est une équation ayant une occurrence de x et $e(y)$ obtenue de $e(x)$ en remplaçant les occurrences de x par y. Dans la règle 7, $e(x)$ est une équation dans laquelle x est le noyau d'un terme et $e(t)$ l'équation obtenue en remplaçant le noyau x par t, et si $e(x)$ est écrite avec $\stackrel{*}{=}$, $e(t)$ est écrite avec $=$. Dans la règle 10, r est soit ε, soit une variable de sorte *queue*. Dans la règle 14, $|\overline{u}| \neq |\overline{v}|$.

Deuxième phase Cette phase s'effectue sur une conjonction c. Selon la forme des équations dans c, les règles suivantes s'appliquent. Lorsqu'il y a une disjonction qui se créée, la disjonction est distribuée pour maintenir la formule sous forme disjonctive $\bigvee_i \exists u_i c'_i$. La phase 1 est relancée sur des conjonctions c'_i.

1. Une équation $[u_1..u_{n+k}]x = y[v_1..v_n]$ avec x, y des variables distinctes, est remplacée par

$$\begin{array}{l} \exists u(y = [u_1..u_{n+k}]u \wedge x = u[v_1..v_n] \wedge queue(u)) \\ \bigvee_{i=0}^{n-1} \left(\begin{array}{l} y = [u_1..u_{k+i}]\varepsilon \wedge x = [v_{n-i+1}..v_n]\varepsilon \\ \wedge v_1 = u_{k+i+1} \wedge \cdots \wedge v_{n-i} = u_{n+k} \end{array} \right) \end{array}$$

   idem pour $[u_1..u_n]x = y[v_1..v_{n+k}]$.
2. Equation $[u_1..u_n]x = x[v_1..v_n]$: Pour chaque couple i, j avec $1 \leq i \neq j \leq n$, on ajoute $(u_i = u_j \wedge v_i = v_j) \vee \neg(u_i = u_j) \vee \neg(v_i = v_j)$. Distribuer pour créer une disjonction de conjonction d'équations et de diséquations. Pour chaque conjonction on peut déterminer si $u_1..u_n$ est une répétition de $u_1..u_k$.
   - dans les cas où $u_1..u_n$ est de la forme $(u_1..u_k)^{n/k}$ avec $u_1..u_k$ primitif, remplacer l'équation $[u_1..u_n]x = x[v_1..v_n]$ par $[u_1..u_k]x \stackrel{p}{=} x[v_1..v_k]$,
   - dans les autres cas, remplacer l'équation $[u_1..u_n]x = x[v_1..v_n]$ par $[u_1..u_n]x \stackrel{p}{=} x[v_1..v_n]$.
3. Equation $[\overline{u}]x \stackrel{p}{=} x[\overline{v}]$, avec $|\overline{u}| = |\overline{v}| = n$, remplacée par

$$\bigvee_{k=1}^{n} ([\overline{u}]x \stackrel{*}{=} x[\rho(k, \overline{u})] \wedge [\rho(k, \overline{u})]\varepsilon = [\overline{v}]\varepsilon)$$

4. Deux équations $\stackrel{*}{=}$ sur une même variable x, avec $|\overline{u}| > |\overline{v}|$:

$$\begin{array}{l} [\overline{u}]x \stackrel{*}{=} x[\rho(k, \overline{u})] \wedge [\overline{v}]x \stackrel{*}{=} x[\rho(l, \overline{v})] \implies \\ \bigvee_{i \in I}(x = [\overline{u}^i u_1..u_k]\varepsilon \wedge [\overline{v}]x \stackrel{*}{=} x[\rho(l, \overline{v})]) \end{array}$$

   Ici $I = \{0\}$ si $k \geq |\overline{v}|$ et $I = \{0, 1\}$ sinon.
5. Deux équations $\stackrel{*}{=}$ sur une même variable x, avec $|\overline{u}| = |\overline{v}|$ et $k \neq l$:

$$[\overline{u}]x \stackrel{*}{=} x[\rho(k, \overline{u})] \wedge [\overline{v}]x \stackrel{*}{=} x[\rho(l, \overline{v})] \implies false$$

6. Deux équations $\stackrel{*}{=}$ sur une même variable x, avec $|\overline{u}| = |\overline{v}|$:

$$\begin{array}{l} [\overline{u}]x \stackrel{*}{=} x[\rho(k, \overline{u})] \wedge [\overline{v}]x \stackrel{*}{=} x[\rho(k, \overline{v})] \implies \\ \left(\begin{array}{l} (\bigwedge_{i=1}^{k} u_i = v_i \wedge x = [u_1...u_k]\varepsilon) \vee \\ (\bigwedge_{i=1}^{|\overline{u}|} u_i = v_i \wedge [\overline{u}]x \stackrel{*}{=} x[\rho(k, \overline{u})]) \end{array} \right) \end{array}$$

Propriété 4 *Soit c une conjonction de contraintes élémentaires complètement typée. L'application des 2 phases autant que possible sur c se termine et produit*

une formule équivalente qui est soit false soit de la forme $\bigvee_i \exists \overline{x_i}(c_i \wedge \bigwedge_j \neg e_{ij})$*, où les* c_i *sont des conjonctions résolues, les* e_{ij} *sont des équations entre deux variables et chaque variable de* $\overline{x_i}$ *est accessible depuis une variable libre dans* c_i*.*

Exemple 1 *Transformer la conjonction (*$\stackrel{i}{\equiv}$ *signifie la transformation par la phase i) :*

$$
\begin{array}{ll}
 & \begin{pmatrix} x=[u]y \wedge x=z[v] \wedge u=f(v) \wedge queue(x) \\ \wedge queue(y) \wedge queue(z) \wedge arbre(u) \wedge arbre(v) \end{pmatrix} \\
\stackrel{1}{\equiv} & \begin{pmatrix} x=[u]y \wedge [u]y=z[v] \wedge u=f(v) \wedge queue(x) \\ \wedge queue(y) \wedge queue(z) \wedge arbre(u) \wedge arbre(v) \end{pmatrix} \\
\stackrel{2}{\equiv} & \begin{pmatrix} x=[u]y \wedge u=f(v) \wedge queue(x) \\ \wedge queue(y) \wedge queue(z) \wedge arbre(u) \wedge arbre(v) \\ \wedge \begin{pmatrix} \exists w(y=w[v] \wedge z=[u]w \wedge queue(w)) \\ \vee (y=\varepsilon \wedge z=\varepsilon \wedge u=v) \end{pmatrix} \end{pmatrix} \\
\equiv & \begin{pmatrix} \exists w \begin{pmatrix} x=[u]y \wedge u=f(v) \wedge y=w[v] \wedge z=[u]w \\ \wedge queue(w) \wedge queue(x) \wedge queue(y) \\ \wedge queue(z) \wedge arbre(u) \wedge arbre(v) \end{pmatrix} \\ \vee \begin{pmatrix} x=[u]y \wedge u=fv \wedge y=\varepsilon \wedge z=\varepsilon \wedge u=v \\ \wedge queue(x) \wedge queue(y) \wedge queue(z) \\ \wedge arbre(u) \wedge arbre(v) \end{pmatrix} \end{pmatrix} \\
\stackrel{1}{\equiv} & \begin{pmatrix} \exists w \begin{pmatrix} x=[u]w[v] \wedge u=fv \wedge y=w[v] \wedge z=[u]w \\ \wedge queue(w) \wedge queue(x) \wedge queue(y) \\ \wedge queue(z) \wedge arbre(u) \wedge arbre(v) \end{pmatrix} \\ \vee \begin{pmatrix} x=[u]\varepsilon \wedge u=v \wedge y=\varepsilon \wedge z=\varepsilon \wedge v=f(v) \\ \wedge queue(x) \wedge queue(y) \wedge queue(z) \\ \wedge arbre(u) \wedge arbre(v) \end{pmatrix} \end{pmatrix}
\end{array}
$$

4 Algorithme de décision

Nous présentons dans cette section un algorithme pour décider la valeur de vérité d'une formule F du premier ordre. L'algorithme consiste à transformer F en forme prénexe QF', à transformer la matrice F' sans quantificateur en une disjonction de formules basiques et à éliminer successivement les quantificateurs de Q du plus interne au plus externe. L'algorithme est détaillé dans la sous-section 4.1 et les techniques d'élimination de quantificateur sont présentées dans 4.2. Nous discutons la décidabilité dans la sous-section 4.3.

4.1 Algorithme

Soit F une formule du premier-ordred. Pour chaque variable libre x nous ajoutons à F la formule $queue(x) \vee arbre(x)$ et chaque sous-formule $\exists x F'$ est changée en $\exists x(F' \wedge (queue(x) \vee arbre(x))$. Les contraintes sont mises sous forme de contraintes élémentaires, des nouvelles variables sont introduites si nécessaires et sont quantifiées existentiellement, avec une contrainte de sorte respective. La formule obtenue est ensuite mise sous forme prénexe, c'est-à-dire tous les quantificateurs sont mis au début de la formule (éventuellement en renommant des variables quantifiées), puis le reste de la formule est mis en forme normale disjonctive $\bigvee(c \wedge \bigwedge \neg c_i)$, où les c et c_i sont des conjonctions de contraintes élémentaires. Il est claire que ces transformations préservent l'équivalence de formules.

Pour chaque sous-formule $c \wedge \bigwedge \neg c_i$, appliquer l'algorithme de transformation en forme résolue sur c et les c_i. Nous obtenons une formule sous forme

$$\bigvee \exists \overline{u}(c' \wedge \bigwedge_{i \in I'} \neg \exists \overline{u'_i}(c'_i \wedge \bigwedge_j \neg e_{ij})).$$

Cette formule est transformée en la formule :

$$\bigvee \exists \overline{u}(c' \wedge \bigwedge_{i \in I'} (\neg \exists \overline{u'_i} c'_i \vee \bigvee_j \exists \overline{u'_i}(c'_i \wedge e_{ij}))).$$

En faisant des distribution des $\vee$ sur des $\wedge$, la formule est transformée en une formule de la forme

$$\bigvee \exists \overline{u}(c \wedge \bigwedge_{i \in I} \neg \exists \overline{u_i} c_i). \quad (1)$$

En mettant les conjonctions c en forme résolue pour celles qui ne le sont pas encore, on obtient une formule toujours de la forme (1) et où les conjonctions c et c_i sont sous forme résolue, où chaque variable de $\overline{u_i}$ est accessible dans c_i depuis une variable libre. En gardant dans les $\overline{u}$ seulement les variables accessibles dans c depuis une variable libre, les autres variables de $\overline{u}$ étant remontées dans la partie des quantificateurs de F, nous avons une formule de la forme $Q(F1 \vee \ldots \vee F_n)$, où Q dénote la partie des quantificateurs et les F_i sont des formules basiques. Nous procédons ensuite l'élimination des quantificateurs de Q.

Considérons le quantificateur le plus interne de la formule est supposons que ce soit un quantificateur existentiel. C'est-à-dire la formule est $Q\exists x(F_1 \vee \ldots \vee F_n)$, où Q représente le reste des quantificateur. Cette formule est équivalente à $Q(\exists x F_1 \vee \ldots \vee \exists x F_n)$. Le travail devient à transformer chaque formule $\exists x F_i$ en disjonction de formules basique. Il fera l'objet de la sous-section 4.2.

Si le quantificateur le plus interne est un quantificateur universel, la formule est $Q\forall x(F_1 \vee \ldots \vee F_n)$, qui devient $Q\neg\exists x\neg(F_1 \vee \ldots \vee F_n)$ donc $Q\neg\exists x(\neg F_1 \wedge \ldots \wedge \neg F_n)$. Chaque formule F_i est une formule basique donc de la forme $\exists \overline{u}(c \wedge \bigwedge \neg \exists \overline{u_i} c_i)$. Elle est transformée en une conjonction $\exists \overline{u} c \wedge \bigwedge \neg \exists \overline{u} \exists \overline{u_i}(c \wedge c_i)$. Nous transformons ensuite $(\neg F_1 \wedge \ldots \wedge \neg F_n)$ en forme normale disjonctive et éliminons des quantificateurs existentielles introduits comme précédent. Après l'élimination du quantificateur $\exists x$, la négation du résultat est ensuite mise en forme de disjonction de formules basiques comme précédent.

4.2 Elimination de quantificateur

Soit x une variable et b une formule basique. L'objectif de l'élimination du quantificateur $\exists x$ est de transformer $\exists x b$ en une disjonction de formules basiques. Nous présentons cette élimination en deux parties, dépendant de la forme de b : (1) lorsque b ne contient pas de négation, et (2) lorsque b contient des négations.

4.2.1 Elimination dans une conjonction résolue

Lorsque la formule basique b ne contient pas de négation, il est de la forme $\exists \overline{u} c$, avec c une conjonction résolue. S'il existe une variable libre y telle que $x \in Acc_c(y)$, alors $\exists x \exists \overline{u} c$ est déjà une formule basique. Si x n'est pas accessible depuis une variable libre, la formule $\exists x \exists \overline{u} c$ peut s'écrire en $\exists \overline{u'}(c' \wedge \exists x \overline{v} c_x)$, où $\overline{v}$ est le vecteur des variables de $\overline{u}$ qui sont accessibles depuis x et non accessibles depuis aucune autre variable libre, c_x est la conjonction des contraintes élémentaires qui font intervenir des variables de $x\overline{v}$ et c' la conjonction des autres contraintes de c. L'élimination de x dans $\exists x \exists \overline{u} c$ retourne $\exists \overline{u'} c'$.

Exemple 2 *Eliminons $\exists x$ dans $\exists x \exists u v_1 v_2([v_1 v_2]x = x[v_1 v_2] \wedge v_1 = f(u) \wedge y = f(u) \wedge queue(x) \wedge arbre(v_1) \wedge queue(v_2) \wedge arbre(y) \wedge queue(u))$. Dans cette formule x n'est pas accessible depuis une autre variable libre, u est accessible depuis x, y et v_1, v_2 sont accessibles depuis x. La formule se réécrit en :*

$$\exists u \begin{pmatrix} y = f(u) \wedge arbre(y) \wedge queue(u) \wedge \\ \exists x v_1 v_2 \begin{pmatrix} [v_1 v_2]x = x[v_1 v_2] \wedge v_1 = f(u) \wedge \\ queue(x) \wedge arbre(v_1) \wedge queue(v_2) \end{pmatrix} \end{pmatrix}$$

et le résultat de l'élimination est

$$\exists u(y = f(u) \wedge arbre(y) \wedge queue(u)).$$

4.2.2 Elimination dans une formule basique

Soient x une variable et b une formule basique. Nous allons transformer $\exists x b$ en une disjonction de formules basiques. Si x n'a pas d'occurrence dans b le résultat sera b. Supposons que x ait des occurrences dans b. Soit b la formule basique $\exists \overline{u}(c \wedge \bigwedge_{i \in I} \neg \exists \overline{u_i} c_i)$. S'il existe une autre variable libre y de b telle que $x \in Acc_c(y)$, alors $\exists x b$ est une formule basique. Dans le cas contraire, où x n'est accessible dans c depuis aucune autre variable libre, tous les cas possibles pour x sont listés suivant avec les transformations correspondantes. Note : puisque les contraintes de type *queue* et *arbre* respectent le type des symboles de fonction, pour rendre les formules plus visibles dans les exemples, nous omettons ces contraintes dans les cas où cela ne change pas la sémantique des formules.

(a) x est de sorte arbre et x a des occurrences dans des équations de c Du fait que x a des occurrences dans c mais x n'est accessible dans c depuis aucune autre variable libre, x doit être le membre gauche d'une équation $x = t$. L'ensemble $Acc_c(x)$ est réparti en deux sous-ensemles disjoints $Acc^a_c(x)$ des variables de sorte *arbre* et $Acc^q_c(x)$ des variables de sorte *queue*. Soit $A^q_c(x)$ l'ensemble de tous les variables accessibles dans c depuis une variable de $Acc^q_c(x)$. Notons que $x \notin A^q_c(x)$, car sinon il doit exister une variable w de sorte *queue* dans $Acc^q_c(x)$ telle que $w \in Acc_c(w)$, ce qui contrarie le fait que c est résolue. Nous pouvons donc calculer un vecteur $\overline{v}$ des variables de sorte *arbre* de $\overline{u}$ qui sont dans $Acc^a_c(x) \backslash A^q_c(x)$ et qui sont des membres gauches d'équations de c. Soit c_x la conjonction de ces équations et des contraintes *arbre* concernant ces variables et soit c' le reste des contraintes de c. Soit $\overline{u'}$ le vecteur des variables de $\overline{u}$ qui ne sont pas dans $\overline{v}$. Notons que ni x ni aucune variable de $\overline{v}$ n'a d'occurrence dans c', car si une variable $v \in x\overline{v}$ est dans c', puisque c est résolue, v ne peut pas être un représentant de c', on a donc $v \in A^q_c(x)$, ce qui est contraire à la définition de $\overline{v}$. La formule $\exists x b$ se réécrit en :

$$\exists \overline{u'}(c' \wedge \exists x \overline{v} c_x \wedge \bigwedge_{i \in I} \neg \exists \overline{u_i} c_i)$$

pour être transformée en :

$$\exists \overline{u'}(c' \wedge \bigwedge_{i \in I} \neg \exists \overline{u_i} \exists x \overline{v}(c_x \wedge c_i)).$$

La procédure présentée en 4.2.1 est appliquée dans les négations. Le quantificateur $\exists x$ est donc éliminé. Pour rendre le résultat en forme basique il faudra éventuellement éliminer d'autres quantificateurs de $\exists \overline{u'}$.

Exemple 3 *Eliminer $\exists x_1$:*

$$\exists x_1 \exists y w \begin{pmatrix} x_1 = f(y, v) \wedge y = g(z) \wedge z = g(y) \wedge w = g(x_1) \\ \wedge [z]v = v[z] \wedge queue(u) \wedge queue(v) \wedge arbre(x_1) \\ \wedge arbre(x_2) \wedge arbre(y) \wedge arbre(z) \wedge arbre(w) \\ \wedge \neg(u = [z]v)) \wedge \neg(x_2 = g(x_1)) \end{pmatrix}$$

Ici $Acc^a_c(x_1) = \{y, z, w\}$, $Acc^q_c(x_1) = \{v\}$ et $A^q_c(x_1) = \{y, z\}$. La formule est transformée en

$$\exists y \begin{pmatrix} y = g(z) \wedge z = g(y) \wedge [z]v = v[z] \wedge queue(u) \\ \wedge queue(v) \wedge arbre(x_2) \wedge arbre(y) \wedge arbre(z) \\ \wedge \neg \exists x_1 w \begin{pmatrix} x_1 = f(y, v) \wedge w = g(x_1) \wedge u = [z]v) \\ \wedge arbre(x_1) \wedge arbre(w) \end{pmatrix} \\ \wedge \neg \exists x_1 w \begin{pmatrix} x_1 = f(y, v) \wedge w = g(x_1) \wedge x_2 = g(x_1) \\ \wedge arbre(x_1) \wedge arbre(w) \end{pmatrix} \end{pmatrix}$$

puis en

$$\exists y \begin{pmatrix} y=g(z) \wedge z=g(y) \wedge [z]v=v[z] \wedge queue(u) \\ \wedge queue(v) \wedge arbre(x_2) \wedge arbre(y) \wedge arbre(z) \\ \wedge \neg(u=[z]v) \\ \wedge \neg\exists x_1 w \begin{pmatrix} x_1=f(y,v) \wedge w=g(x_1) \wedge x_2=g(x_1) \\ \wedge arbre(x_1) \wedge arbre(w) \end{pmatrix} \end{pmatrix}$$

Cette formule est une formule basique.

(b) x est de sorte arbre et x n'a pas d'occurrence dans les équations de c L'ensemble I est séparé en deux sous-ensembles disjoints I_1 et I_2, où I_1 est l'ensemble des indices i tels que c_i contient au moins une occurrence de x. Soit c' la conjonction des contraintes de c à l'exception de $arbre(x)$. L'élimination de x dans $\exists xb$ retourne :

$$\exists \overline{u}(c' \wedge \bigwedge_{i \in I_2} \neg\exists \overline{u_i} c_i).$$

Cette formule est une formule basique.

Exemple 4 *Eliminant $\exists x_2$ dans la dernière formule de l'exemple 3, on obtient la formule basique :*

$$\exists y \begin{pmatrix} y=g(z) \wedge z=g(y) \wedge [z]v=v[z] \wedge queue(u) \\ \wedge queue(v) \wedge arbre(y) \wedge arbre(z) \wedge \neg(u=[z]v) \end{pmatrix}$$

(c) x est de sorte queue et une équation de la forme $x = t$ est dans c Du fait que x n'est pas accessible depuis une autre variable libre, x n'a pas d'autres occurrences dans les autres équations de c. Soit c' la conjonction des contraintes de c à l'exception de $x = t$ et de $queue(x)$. La formule $\exists xb$ est transformée en :

$$\exists \overline{u}(c' \wedge \bigwedge_{i \in I} \neg\exists x \overline{u_i}(x = t \wedge queue(x) \wedge c_i)).$$

La procédure présentée en 4.2.1 est appliquée dans les négations. Le quantificateur $\exists x$ est donc éliminé. Pour rendre le résultat en forme basique il faut éventuellement éliminer d'autres quantificateurs de $\exists \overline{u}$.

Exemple 5 *Transformation pour éliminer $\exists x$:*

$$\begin{array}{ll} & \exists x \exists y v(x=[v]y \wedge \neg(x=\varepsilon) \wedge \neg\exists u z(x=[u]z)) \\ \equiv & \exists y v(\neg\exists x(x=[v]y \wedge x=\varepsilon) \wedge \neg\exists x u z(x=[v]y \wedge x=[u]z)) \\ \equiv & \exists y v(\neg false \wedge \neg\exists x u z(x=[u]z \wedge u=v \wedge z=y)) \\ \equiv & \exists y v(\neg true) \\ \equiv & false \end{array}$$

(d) x est de sorte queue et une équation de la forme $[\overline{u}]x = x[\rho(k, \overline{u})]$ est dans c Nous pouvons supposer ici et dans le cas (e) que si x a une occurrence dans une conjonction c_i alors x a une occurrence dans une équation de c_i. En effet, si x n'est dans aucune équation de c_i et seule la contrainte $queue(x)$ est dans c_i, nous pouvons la supprimer grâce à l'équivalence $queue(x) \wedge \neg\exists \overline{u_i}(c'_i \wedge queue(x)) \equiv queue(x) \wedge \neg\exists \overline{u_i} c'_i$. Du fait que x n'est pas accessible depuis une autre variable libre, x n'a pas d'autres occurrences dans les autres équations de c. Les cas possibles pour x sont :

d1. Dans chaque conjonction c_i qui contient une occurrence de x, il existe une variable libre y_i telle que $x \in Acc_{c_i}(y_i)$, ou c_i contient une équation de l'une des formes $x=\varepsilon$, $x = y$ et $x = [\overline{v}]y[\overline{v'}]$ avec y libre. L'élimination de $\exists x$ dans $\exists xb$ retourne :

$$\exists \overline{u}(c' \wedge \bigwedge_{i \in I'} \neg\exists \overline{u_i} c_i)$$

où c' est la conjonction des contraintes de c à l'exception de $[\overline{u}]x = x[\rho(k, \overline{u})]$ et de $queue(x)$, I' est le sous-ensemble maximal de I tel que pour tout $i \in I'$, c_i n'a pas d'occurrence de x. Cette formule est une formule basique.

Exemple 6 *Eliminant $\exists v$ dans la formule résultante de l'exemple 4 on obtient la formule basique :*

$$\exists y \begin{pmatrix} y=g(z) \wedge z=g(y) \wedge queue(u) \\ \wedge arbre(y) \wedge arbre(z) \end{pmatrix}$$

d2. Il existe une conjonction c_i dans laquelle x n'est pas accessible depuis une autre variable libre et qui contient une équation de la forme $[\overline{v}]x = x[\rho(l, \overline{v})]$. En utilisant la procédure de résolution de contraintes d'arbres [4, 6], nous montrons comment supprimer l'équation $[\overline{v}]x = x[\rho(l, \overline{v})]$. La formule $\exists xb$ est transformée en :

$$\exists x \overline{u}(c \wedge \neg\exists \overline{u_i}(c \wedge c_i) \wedge \bigwedge_{j \in I, j \neq i} \neg\exists \overline{u_j} c_j).$$

La conjonction $c \wedge c_i$ est mise sous forme résolue. Dans cette procédure, si $|\overline{u}| \neq |\overline{v}|$ ou $k \neq l$, les deux équations $[\overline{u}]x = x[\rho(k, \overline{u})]$ et $[\overline{v}]x = x[\rho(l, \overline{v})]$ sont supprimées et remplacées par des équations de la forme $x = t$, ce cas fait l'objet de l'item d3. suivant. Sinon, dans la règle 6 de la deuxième phase, l'équation $[\overline{u}]x = x[\rho(k, \overline{u})]$ est gardée. Les équations recopiées de c seront restaurées puis supprimées, d'après la procédure dans [4, 6]. L'équation $[\overline{v}]x = x[\rho(l, \overline{v})]$ est ainsi supprimée.

Exemple 7 *Eliminer $\exists x$:*

$$\begin{array}{ll} & \exists x([u]x = x[u] \wedge \neg\exists y(x=[u]y) \wedge \neg([v]x = x[v])) \\ \equiv & \exists x \begin{pmatrix} [u]x = x[u] \wedge \neg\exists y(x=[u]y) \wedge \\ \neg([u]x = x[u] \wedge [v]x = x[v]) \end{pmatrix} \\ \equiv & \exists x \begin{pmatrix} [u]x = x[u] \wedge \neg\exists y(x=[u]y) \wedge \\ \neg(x=\varepsilon) \wedge \neg([u]x = x[u] \wedge u=v) \end{pmatrix} \\ \equiv & \exists x \begin{pmatrix} [u]x = x[u] \wedge \neg\exists y(x=[u]y) \wedge \\ \neg(x=\varepsilon) \wedge \neg(u=v) \end{pmatrix} \end{array}$$

L'équation $[v]x = x[v]$ est donc supprimée. La suite se trouve dans l'exemple 8.

d3. Il existe une conjonction c_i dans laquelle x n'est pas accessible depuis une autre variable libre et qui contient une équation de la forme $x = [\overline{v}]y[\overline{v'}]$, avec $y \in \overline{u_i}$. Ici $\overline{v}\overline{v'}$ doit être non vide car sinon l'équation devient $x = y$, ce qui entraîne que $no(x) > no(y)$ contradictoire avec le fait que $y \in \overline{u_i}$. Si $\overline{v}$ n'est pas vide, soit a le plus petit nombre tel que $|\overline{u}|a \geq |\overline{v}|$. L'équation $[\overline{u}]x = x[\rho(k, \overline{u})]$, avec $\overline{u} = u_1...u_k...u_n$ est remplacée par la disjonction

$$\bigvee_{j=0}^{a-1} (x = [\overline{u}^j u_1..u_k]\varepsilon) \vee \exists z(x = [\overline{u}^a]z \wedge [\overline{u}]z = z[\rho(k, \overline{u})])$$

Sinon soit a le plus petit nombre tel que $|\overline{u}|a \geq |\overline{v'}|$. L'équation $[\overline{u}]x = x[\rho(k, \overline{u})]$ est remplacée par la disjonction

$$\bigvee_{j=0}^{a-1} (x = [\overline{u}^j u_1..u_k]\varepsilon) \vee \exists z(x = z[\overline{u}^a] \wedge [\overline{u}]z = z[\rho(k, \overline{u})])$$

La distribution des $\vee$ sur des $\wedge$ est effectuée et on se retrouve dans le cas (c). La variable z reste à éliminer mais le fait d'associer à x une liste de longueur supérieure à $|\overline{v}|$ (ou $|\overline{v'}|$) permet de ne pas créer une équation de la forme $z = t$.

Exemple 8 *Suite de l'exemple 7 :*

$$\begin{array}{l} \exists x([u]x = x[u] \wedge \neg\exists y(x = [u]y) \wedge \neg(x = \varepsilon) \wedge \neg(u = v)) \\ \equiv \begin{pmatrix} \exists x(x = \varepsilon \wedge \neg\exists y(x = [u]y) \wedge \neg(x = \varepsilon) \wedge \neg(u = v)) \\ \vee \\ \exists xz \begin{pmatrix} x = [u]z \wedge [u]z = z[u] \wedge \neg\exists y(x = [u]y) \wedge \\ \neg(x = \varepsilon) \wedge \neg(u = v) \end{pmatrix} \end{pmatrix} \end{array}$$

Elimination de $\exists x$ dans la première formule de la disjonction :

$$\begin{array}{ll} & \exists x(x = \varepsilon \wedge \neg\exists y(x = [u]y) \wedge \neg(x = \varepsilon) \wedge \neg(u = v)) \\ \equiv & \begin{pmatrix} \neg\exists xy(x = \varepsilon \wedge x = [u]y) \wedge \neg\exists x(x = \varepsilon \wedge x = \varepsilon) \\ \wedge\neg\exists x(x = \varepsilon \wedge u = v) \end{pmatrix} \\ \equiv & \neg \mathit{false} \wedge \neg \mathit{true} \wedge \neg(u = v) \\ \equiv & \mathit{false} \end{array}$$

Elimination de $\exists x$ dans la seconde formule :

$$\begin{array}{ll} & \exists xz \begin{pmatrix} x = [u]z \wedge [u]z = z[u] \wedge \neg\exists y(x = [u]y) \wedge \\ \neg(x = \varepsilon) \wedge \neg(u = v) \end{pmatrix} \\ \equiv & \exists z \begin{pmatrix} [u]z = z[u] \wedge \neg\exists xy(x = [u]z \wedge x = [u]y) \wedge \\ \neg\exists x(x = [u]z \wedge x = \varepsilon) \wedge \\ \neg\exists x(x = [u]z \wedge u = v) \end{pmatrix} \\ \equiv & \exists z \begin{pmatrix} [u]z = z[u] \wedge \neg\exists xy(x = [u]z \wedge y = z) \wedge \\ \neg \mathit{false} \wedge \neg(u = v) \end{pmatrix} \\ \equiv & \exists z([u]z = z[u] \wedge \neg \mathit{true} \wedge \neg(u = v)) \\ \equiv & \mathit{false} \end{array}$$

Le résultat est donc la formule false.

(e) x est de sorte queue et x n'a pas d'occurrence dans les équations de c Les cas possibles sont :

e1. Pour chaque conjonction c_i qui contient une occurrence de x, soit x est accessible dans c_i depuis une autre variable libre, soit c_i contient une équation de l'une des formes $x = y$, $x = \varepsilon$, $[\overline{v}]x = x[\rho(k, \overline{v'})]$ et $x = [\overline{v}]y[\overline{v'}]$, avec y une variable libre. La formule $\exists xb$ est transformée en :

$$\exists \overline{u}(c' \wedge \bigwedge_{i \in I'} \neg\exists \overline{u_i} c_i)$$

où c' est la conjonction des contraintes de c à l'exception de $queue(x)$, I' est le sous-emsemble maximal de I tel que pour tout $i \in I'$, c_i n'a pas d'occurrence de x. Cette formule est une formule basique.

Exemple 9 *Eliminer $\exists x$:*

$$\begin{array}{ll} & \exists x \exists u \begin{pmatrix} y = f(u) \wedge queue(x) \wedge \\ \neg\exists v(u = [v]z) \wedge \exists v([v]x = x[v]) \end{pmatrix} \\ \equiv & \exists u(y = f(u) \wedge \neg\exists v(u = [v]z)) \end{array}$$

e2. Il existe une conjonction c_i dans laquelle x n'est pas accessible depuis une autre variable libre et c_i contient une équation $x = [\overline{u}]y[\overline{u'}]$ où les variables de $y\overline{u}\overline{u'}$ sont tous quantifiées. Soit $n = |\overline{u}| + |\overline{u'}|$. Puisque $queue(x)$ est dans c, en conservant l'équivalence nous ajoutons dans c la disjonction

$$\begin{array}{l} (x = \varepsilon) \vee \bigvee_{1 \leq i < n} \exists v_1..v_i(x = [v_1..v_i]\varepsilon) \vee \\ \exists y v_1...v_n(x = [v_1..v_{|\overline{u}|}]y[v_{|\overline{u}|+1}..v_n] \wedge queue(y)) \end{array}$$

La distribution des $\vee$ sur des $\wedge$ est effectuée et on se retrouve dans le cas (c). Les variables $y\overline{u}\overline{u'}$ restent à éliminer mais le fait d'associer à x une liste soit de longueur déterminée soit de longueur supérieure à $|\overline{u}| + |\overline{u'}|$ permet de supprimer des négations de ce cas.

Exemple 10 *Transformations pour éliminer $\exists x$:*

$$\begin{array}{ll} & \exists x(queue(x) \wedge \neg(x = \varepsilon) \wedge \neg\exists uz(x = [u]z)) \\ \equiv & \begin{pmatrix} \exists x(x = \varepsilon \wedge \neg(x = \varepsilon) \wedge \neg\exists uz(x = [u]z)) \vee \\ \exists xyv(x = [v]y \wedge \neg(x = \varepsilon) \wedge \neg\exists uz(x = [u]z)) \end{pmatrix} \end{array}$$

D'après l'exemple 5, l'élimination de $\exists x$ dans les deux formules de la disjonction retourne false. Le résultat est donc la formule false.

4.3 Décidabilité

Dans la procédure d'élimination de quantificateur nous n'introduisons pas de nouvelle variable libre. Les cas d2, d3 et e2, où des nouvelles variables quantifiées sont ajoutées se ramènent toujours à un autre cas, où des variables quantifiées sont éliminées sans ajouter

de nouvelles. La procédure va s'arrêter donc après un temps fini. La correction et la terminaison de cette procédure se résument dans le résultat suivant.

Théorème 1 *Soit x une variable et b une formule basique. La procédure d'élimination de quantificateur transforme $\exists xb$ après un temps fini en une disjonction de formules basiques, équivalente à $\exists xb$ dans l'algèbre des arbres finis ou infinis avec des queues.*

Si l'algorithme s'applique sur une formule close, tous les quantificateurs seront éliminés par cette procédure. Nous obtiendrons soit la valeur de vérité *true* (le résultat est une disjonction contenant une seule formule basique *true*) soit la valeur de vérité *false* (le résultat est une disjonction vide, donc *false*). On conclut donc :

Corollaire 1 *La théorie du premier ordre de l'algèbre des arbres finis ou infinis avec queues est complète et décidable.*

5 Conclusion

Dans ce papier nous avons présenté l'algèbre des arbres finis ou infinis étendus avec des queues. L'ensemble des symboles de fonction de la signature est infini. Nous avons présenté un algorithme d'élimination de quantificateurs dans cette algèbre. Cette algorithme permet de décider la valeur de vérité des formules du premier ordre closes dans cette algèbre. Il montre également que la théorie du premier ordre de l'algèbre est complète et décidable.

Un travail restant à faire est de formuler une axiomatisation de la théorie du premier ordre des arbres finis ou infinis avec des queues. Une solution, comme en [13], est de rassembler les propriétés nécessaires pour assurer la correction de l'algorithme. Nous continuons à travailler sur cet aspect à fin de caractériser une formulation simple et concise. D'un autre côté, en se basant sur cet algorithme de décision, nous étudions des adaptations pour répondre à l'objectif de résoudre des contraintes du premier ordre dans cette algèbre.

Remerciements Nous remercions Alain Colmerauer pour les nombreuses discussions sur la théorie des arbres avec queues ainsi que ses remarques et ses conseils sur notre travail.

Références

[1] K.L. Clark, Negation as failure, in Logic and Databases, Ed. H. Gallaire and J. Minker, Plenum Press, 293-322, 1978.

[2] A. Colmerauer, An introduction to Prolog III, Commun. ACM 33(7), 69-90, 1990.

[3] H. Comon, Résolution de contraintes dans des algèbres de termes. Rapport d'habilitation, Université de Paris Sud, 1991.

[4] T-B-H. Dao, Résolution de contraintes du premier ordre dans la théorie des arbres finis ou infinis, Thèse d'Informatique, Université Aix-Marseille II, 2000.

[5] K. Djelloul, T-B-H. Dao, Extentions into trees of first-order theories, AISC06, 53-67, 2006.

[6] K. Djelloul, T-B-H. Dao, T. Frühwirth, Theory of finite or infinite trees revisisted, TPLP, 2008.

[7] J. Jaffar, M. Maher, K. Marriott, P. Stuckey, The semantics of constraint logic programs, The Journal of Logic Programming 37(1998), 1-46.

[8] K. Kunen, Negation in logic programming, Journal of Logic Programming, 4, 289-308, 1987.

[9] M. Lothaire, Combinatorics on words, Encyclopedia of Mathematics, Vol 17, Addison-Wesley, 1983. Reprinted in 1997 by Cambridge University Press, in Cambridge Mathematical Library.

[10] M. J. Maher, Complete axiomatization of the algebra of finite, rational and infinite trees, Technical report, IBM-T.J. Watson Research Center, 1988.

[11] A. Malcev, Axiomatizable classes of locally free algebras of various types, In The Metamathematics of Algebraic Systems : Collected Paper, chapter 23, 262-281, 1971.

[12] G. Nelson, D.C. Oppen, Simplification by cooperating decision procedures, ACM Transactions on Programming Languages and Systems 1(2), 245-257, 1979.

[13] T. Rybina, A. Voronkov, A Decision Procedure for Term Algebras with Queues, ACM Transactions on Computational Logic, Vol. 2, No. 2, April 2001, pages 155-181.

Cohérences basées sur les valeurs en échec

Christophe Lecoutre **Olivier Roussel**

Université Lille-Nord de France, Artois, F-62307 Lens
CRIL - CNRS UMR 8188, F-62307 Lens
F-62307 Lens
{lecoutre,roussel}@cril.fr

Résumé

Dans le domaine de la satisfaction de contraintes, les inférences se basent en général sur certaines propriétés des réseaux de contraintes, appelées cohérences. Celles-ci permettent l'identification d'instanciations incohérentes, ou nogoods. Deux familles principales de cohérences ont été étudiées jusqu'ici : celles qui permettent de raisonner à partir des variables telles que la (i, j)-cohérence et celles qui permettent de raisonner à partir des contraintes telles que la (i, j)-cohérence relationnelle. Dans cet article, nous introduisons une nouvelle famille de cohérences basées sur le concept de valeur en échec (valeur éliminée au cours de la recherche). Cette famille est orthogonale aux précédentes.

Abstract

In constraint satisfaction, basic inferences rely on some properties of constraint networks, called consistencies, that allow the identification of inconsistent instantiations or nogoods. Two main families of consistencies have been introduced so far : those that permit to reason from variables such as (i, j)-consistency and those that permit to reason from constraints such as relational (i, j)-consistency. This paper introduces a new family of consistencies based on the concept of failed value (a value pruned during search). This family is orthogonal to previous ones.

1 Introduction

L'utilisateur d'un solveur de contraintes souhaite idéalement que le système soit suffisamment robuste et intelligent pour identifier et exploiter automatiquement toutes les propriétés d'une instance à résoudre. Typiquement, ces propriétés dépendent de la structure de l'instance et permettent de la résoudre plus facilement. Diverses approches sont explorées pour atteindre ce but : l'utilisation de cohérences fortes, les heuristiques adaptatives, l'apprentissage de nogoods et l'identification de symétries. Ces techniques permettent d'explorer efficacement l'espace de recherche en glanant de précieuses informations qui permettent d'élaguer l'arbre de recherche.

Généralement, on utilise une recherche arborescente pour résoudre les problèmes de satisfaction de contraintes (CSP). La recherche arborescente combine une exploration en profondeur d'abord pour instancier les variables avec un mécanisme de retour arrière pour gérer les impasses. Pendant la recherche, certaines valeurs sont identifiées comme incohérentes, i.e. elles ne peuvent figurer dans aucune solution. Nous appelons ces valeurs des valeurs en échec (FV pour Failed Values). On sait [14] que les valeurs en échec apportent une information : étant donnée une instance CSP binaire satisfiable P, pour tout couple (x, a) où x est une variable de P et a une valeur du domaine de x, s'il n'y a pas de solution assignant a à x, alors toute solution assigne nécessairement à au moins une autre variable y une valeur qui est incompatible avec (x, a). Cette propriété est utilisée pour décomposer une instance CSP de manière dynamique et itérée dans [14, 3].

Dans cet article, nous proposons d'utiliser les valeurs en échec de manière différente. En effet, certaines inférences sont possibles en raisonnant localement sur ces valeurs à un coût qui reste intéressant. Plus précisément, à partir des valeurs en échec, nous construisons une nouvelle famille de cohérences locales réduisant les domaines et montrons que cette famille est orthogonale (i.e. incomparable) avec les cohérences habituelles. Elles contribuent à la réduction de l'espace de recherche et permettent accessoirement une détection paresseuse d'une forme généralisée de la relation de substituabilité. Les cohérences basées sur les valeurs en échec s'intègrent aisément à n'importe quel moteur de propagation de contraintes et renforcent le pouvoir de filtrage de l'algorithme de recherche. Ces nouvelles cohérences peuvent contribuer à la mise au point de solveurs

plus robustes.

Après un rappel des définitions, cet article présente la *FV-cohérence d'arc* (AFVC) et un algorithme permettant de maintenir AFVC. Nous comparons ensuite AFVC à la substituabilité ainsi qu'aux cohérences usuelles. Enfin, nous montrons qu'une nouvelle famille de cohérences peut-être construite très naturellement à partir des valeurs en échec.

2 Définitions et notations

Un réseau de contraintes (CN pour *Constraint Network*) P se compose d'un ensemble fini de n variables, noté $vars(P)$, et d'un ensemble fini de e contraintes, noté $cons(P)$. Chaque variable x a un domaine associé noté $dom(x)$ qui contient l'ensemble fini de valeurs qui peuvent être assignées à x. Chaque contrainte c porte sur un ensemble de variables appelé *portée* de c et noté $scp(c)$. Une contrainte est définie par une relation notée $rel(c)$ qui contient l'ensemble des tuples autorisés pour les variables de $scp(c)$. L'*arité* d'une contrainte est le nombre de variables dans sa portée. Une contrainte *binaire* porte sur 2 variables tandis qu'une contrainte *n-aire* porte sur strictement plus de 2 variables. Pour une contrainte binaire c_{xy} telle que $scp(c_{xy}) = \{x, y\}$, on dit que (x, a) et (y, b) sont compatibles ssi $(a, b) \in rel(c_{xy})$. Lorsqu'il n'y a pas de contrainte entre x et y, toute valeur de x est dite compatible avec chaque valeur de y.

On considère donné un réseau initial de contraintes P^{init} et un réseau courant P dérivé de P^{init} en réduisant les domaines des variables. Le domaine initial d'une variable x est noté $dom^{init}(x)$ et le domaine courant est noté $dom^P(x)$ ou simplement $dom(x)$. Pour toute variable x, on a $dom(x) \subseteq dom^{init}(x)$ et on note $P \preceq P^{init}$. Une valeur courante de P est un couple (x, a) avec $x \in vars(P)$ et $a \in dom(x)$. Sans perte de généralité, les réseaux de contraintes considérés dans cet article ne contiendront ni contraintes d'arité 1, ni deux contraintes de même portée (il suffit de normaliser les réseaux [1, 4]).

Une instanciation I d'un ensemble $X = \{x_1, \ldots, x_k\}$ de variables est un ensemble $\{(x_1, a_1), \ldots, (x_k, a_k)\}$ tel que $\forall i, a_i \in dom^{init}(x_i)$; l'ensemble X des variables de I est noté $vars(I)$ et chaque valeur a_i est notée $I[x_i]$. Une instanciation I sur un réseau P est une instanciation d'un ensemble $X \subseteq vars(P)$; elle est complète lorsque $vars(I) = vars(P)$, et partielle dans le cas contraire. I est valide sur P ssi $\forall (x, a) \in I, a \in dom(x)$ $(= dom^P(x))$. Une instanciation I recouvre une contrainte c ssi $scp(c) \subseteq vars(I)$, et elle satisfait une contrainte c avec $scp(c) = (x_1, \ldots, x_r)$ ssi a) I recouvre c et b) le tuple $(a_1, \ldots, a_r)$, tel que $\forall i, a_i = I[x_i]$, est autorisé par c. Une instanciation I sur un réseau P est localement cohérente ssi a) I est valide sur P et b) chaque contrainte de P recouverte par I est satisfaite par I. Si ce n'est pas le cas, I est localement incohérente. Une solution de P est une instanciation complète de P qui est localement cohérente. Une instanciation I sur un réseau P est globalement incohérente (ou également appelé *nogood*) ssi elle ne peut être étendue pour devenir une solution de P. Elle est globalement cohérente dans le cas contraire.

Une solution d'un CN est une assignation d'une valeur à chaque variable de sorte que chaque contrainte soit satisfaite. Un CN est *satisfiable* ssi il possède au moins une solution. Le problème de satisfaction de contraintes (CSP) est le problème NP-dur de déterminer si un réseau donné est satisfiable ou pas. Une instance CSP est déterminée par un CN qui est résolu soit en identifiant une solution, soit en prouvant l'insatisfiabilité. Pour résoudre une instance CSP, une recherche en profondeur d'abord avec retour arrière est souvent utilisée. À chaque étape de la recherche, une variable se voit assigner une valeur. S'ensuit alors un processus de filtrage nommé propagation de contraintes qui permet d'éviter des assignations futures qui seraient incohérentes avec les choix précédents. Typiquement, les algorithmes de propagation de contraintes utilisent des propriétés des réseaux qui permettent d'éliminer des valeurs qui ne peuvent plus apparaître dans une solution. Ces propriétés sont appelées cohérences réduisant les domaines (domain-filtering consistencies) [8, 5].

Nous rappelons brièvement les principales cohérences usuelles. Une instanciation I est un support pour une valeur (x, a) sur une contrainte c portant sur x ssi I est valide, $I[x] = a$ et I satisfait c. Une valeur (x, a) de P est GAC-cohérente (GAC pour Generalized Arc Consistency) ssi il existe un support pour (x, a) sur toute contrainte de P portant sur x. P est GAC-cohérente ssi toute valeur de P est GAC-cohérente. Pour les réseaux binaires, la propriété GAC est nommée AC (cohérence d'arc). Pour tout réseau P, il existe un plus grand sous-réseau de P qui est GAC-cohérent. Ce sous-réseau est la clôture GAC de P que l'on notera $GAC(P)$: cette clôture est équivalente à P et telle que $GAC(P) \preceq P$. Si le domaine d'une variable de P devient vide, P est trivialement insatisfiable (noté $P = \bot$). $P|_{x=a}$ est le réseau obtenu à partir du réseau P en retirant du domaine de x toute valeur $b \neq a$. Une valeur (x, a) de P est SAC-cohérente (SAC pour Singleton Arc Consistency) ssi $GAC(P|_{x=a}) \neq \bot$. P est SAC-cohérente ssi toutes ses valeurs sont SAC-cohérentes.

Une cohérence réduisant les domaines permet d'identifier et d'éliminer des valeurs incohérentes. Un pré-ordre [8] peut être défini sur ces cohérences comme suit. Soient ϕ et ψ deux cohérences. ϕ est plus forte que ψ, noté $\phi \trianglerighteq \psi$, ssi chaque fois que ϕ est vérifiée sur un réseau P, ψ est aussi vérifiée sur P. ϕ est strictement plus forte que ψ, noté $\phi \triangleright \psi$ ssi $\phi \trianglerighteq \psi$ et il existe un réseau P tel que ψ est vérifiée sur P tandis que ϕ ne l'est pas. Quand des cohérences ne peuvent être classées (aucune n'est plus forte que l'autre), elles sont dites *incomparables*. Pour les cohé-

rences usuelles sur les réseaux binaires, on sait que : SAC $\rhd$ MaxRPC $\rhd$ PIC $\rhd$ AC.

3 Cohérences élémentaires basées sur les valeurs en échec

Dans cette section, nous définissons deux nouvelles cohérences élémentaires. La première identifie des nogoods de taille quelconque tandis que la seconde identifie des valeurs incohérentes (nogoods de taille 1). Ces deux cohérences sont basées sur la notion de valeur en échec (FV pour Failed Value). Les valeurs en échec apportent de l'information.

Lemme 1 (dérivé directement de [14])**.** *Si une valeur (x, a) d'un réseau P est globalement incohérente alors chaque solution S de P est telle que $S[x/a]$ viole au moins une contrainte de P impliquant x.*

Démonstration. $S[x/a]$ n'est pas une solution de P puisque (x, a) est globalement incohérente. Cela signifie que au moins une contrainte de P n'est pas satisfaite par $S[x/a]$. Mais nous savons que chaque contrainte c de P qui n'implique pas x est satisfaite par $S[x/a]$ car la restriction de $S[x/a]$ sur $scp(c)$ est exactement la restriction de S sur $scp(c)$. En conséquence, au moins une contrainte de P impliquant x n'est pas satisfaite par $S[x/a]$. □

Définition 1. *Une valeur en échec de P est une valeur d'un réseau $P' \succ P$ qui est globalement incohérente pour P' et qui n'apparaît plus dans P.*

En pratique, une valeur en échec est une valeur retirée d'un réseau parce qu'elle a été prouvée globalement incohérente. Une valeur en échec peut être identifiée en utilisant des techniques d'inférence et/ou de recherche.

Définition 2. *Soit (x, a) une valeur de P. Pour toute contrainte c de P portant sur x, l'ensemble conflit de (x, a) pour c, noté $\chi(c, x, a)$, est l'ensemble des instanciations valides I sur P de $scp(c) \setminus \{x\}$ telles que $I \cup \{(x, a)\}$ ne satisfait pas c. L'ensemble conflit de (x, a) pour P est $\chi(x, a) = \cup_{c \in cons(P) | x \in scp(c)} \chi(c, x, a)$.*

Quel que soit l'ensemble conflit χ, $vars(\chi) = \cup_{I \in \chi} vars(I)$. La figure 1 représente un réseau avec une contrainte binaire entre w et x et une contrainte ternaire entre w, y et z. Dans chacune des figures de cet article, des lignes pleines (respectivement en pointillés) représentent des tuples autorisés (respectivement interdits). L'absence d'arêtes entre deux variables x et y signifie qu'il n'y a aucune contrainte binaire portant sur x et y. Dans cet exemple, $\chi(w, a) = \{\{(x, b)\}, \{(y, a), (z, a)\}\}$ et $\chi(w, c) = \{\{(x, b)\}, \{(x, c)\}, \{(y, c), (z, c)\}\}$.

La définition suivante relie les instanciations aux valeurs en échec.

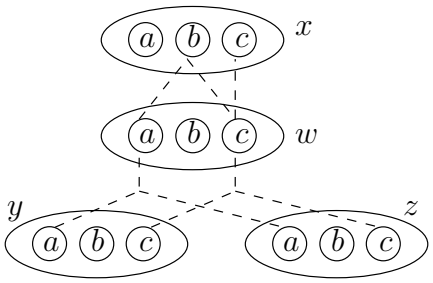


FIG. 1 – Exemple d'ensembles conflit.

Définition 3. *Soit (w, p) une valeur en échec de P et I une instanciation sur P. I recouvre (w, p) ssi $vars(\chi(w, p)) \subseteq vars(I)$. I satisfait (w, p) ssi $\exists J \in \chi(w, p) \mid J \subseteq I$.*

Une valeur en échec satisfaite par une instanciation n'est pas nécessairement recouverte par elle. Cependant, quand une valeur en échec est recouverte par une instanciation et que cette valeur n'est pas satisfaite, un nogood peut être identifié. FVC est utilisé pour Failed Value Consistency.

Définition 4 (FVC)**.** *Une instanciation I valide sur P est FVC-cohérente vis à vis d'une valeur en échec (w, p) de P ssi soit (w, p) n'est pas recouverte par I soit (w, p) est satisfaite par I. I est FVC-cohérente ssi elle est FVC-cohérente vis à vis de toute valeur en échec de P ; elle est dite FVC-incohérente sinon.*

Imaginons que (w, c) de la figure 1 soit une valeur en échec. $I = \{(x, a), (y, c), (z, c)\}$ est une instanciation qui satisfait (w, c) car $\{(y, c), (z, c)\} \in \chi(w, c) \subset I$. $I' = \{(x, a)\}$ ne satisfait pas (w, c) mais est malgré tout FVC-cohérente car (w, c) n'est pas recouverte par I'. $I'' = \{(x, a), (y, a), (z, a)\}$ est FVC-incohérente car (w, c) est à la fois recouverte par I'' et non satisfaite par I''.

Proposition 1. *Toute instanciation FVC-incohérente est globalement incohérente.*

Démonstration. Par définition, quand (w, p) est une valeur en échec de P, il n'y a aucune solution de P contenant (w, p). Pour toute solution S de P, soit $S[w/p]$ l'instanciation complète obtenue à partir de S en y remplaçant la valeur donnée à w par la valeur p. $S[w/p]$ ne peut être une solution car au moins une contrainte portant sur w est nécessairement falsifiée (voir le lemme 1). Quand une instanciation valide I est FVC-incohérente vis à vis de (w, p), c'est qu'il n'y a aucun moyen d'étendre I en une instanciation complète I' telle que $I'[w/p]$ viole au moins une contrainte portant sur w. De ce fait, I est un nogood. □

En d'autres termes, des nogoods peuvent être déduits d'autres nogoods (les valeurs en échec). Ces nogoods déduits peuvent être de taille quelconque comme le montre la figure 2. Dans cet exemple, il y a une valeur en échec

(w, p) et trois contraintes binaires portant sur w. Une instanciation valide de $\{x, y, z\}$ est globalement incohérente si elle contient uniquement des valeurs compatibles avec (w, p). Autrement dit, chaque tuple de $C_x \times C_y \times C_z$ correspond à un nogood (de taille 3).

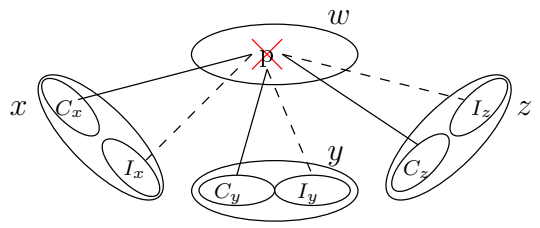


FIG. 2 – Une valeur en échec (w, p), les valeurs compatibles avec (w, p) sont dans C_x, C_y et C_z et les valeurs incompatibles sont dans I_x, I_y et I_z.

Pour les réseaux binaires, une solution directe[1] pour utiliser ces nogoods déduits est de poster, pour chaque valeur en échec (w, p), une contrainte n-aire portant sur $vars(\chi(w, p))$ et qui interdit toute instanciation FVC-incohérente vis à vis de (w, p). Par exemple, dans le cas de la figure 2, on obtient une contrainte ternaire c_{xyz} telle que $rel(c_{xyz}) = dom(x) \times dom(y) \times dom(z) \setminus C_x \times C_y \times C_z$. Il est possible d'utiliser des algorithmes de filtrage efficaces (propagateurs) pour maintenir GAC sur ce type de contraintes. Cependant, cette solution a différents inconvénients : il faut modifier le réseau de contraintes dynamiquement, la généralisation au cas n-aire est complexe et le filtrage est limité aux variables de la contrainte postée.

Cette dernière remarque (filtrage limité) est moins vraie si l'on utilise une cohérence plus forte que GAC telle que SAC. Cependant, les solveurs actuels n'utilisent guère SAC durant la recherche. Intuitivement, on peut utiliser les valeurs en échec pour filtrer davantage en raisonnant sur chaque valeur et les ensembles conflit de chaque valeur en échec. En particulier, on peut définir à partir des valeurs en échec une cohérence réduisant les domaines comme suit. AFVC est utilisé pour Arc Failed Value Consistency.

Définition 5 (AFVC)**.**
- *Une valeur (x, a) de P est AFVC-cohérente vis à vis d'une valeur en échec (w, p) de P ssi (x, a) peut être étendue en une instanciation localement cohérente satisfaisant (w, p).*
- *Une valeur (x, a) de P est AFVC-cohérente ssi (x, a) est AFVC-cohérente vis à vis de toute valeur en échec de P ; elle est dite AFVC-incohérente sinon.*
- *P est AFVC-cohérente ssi chaque valeur de P est AFVC-cohérente.*

Dans le premier point de la définition, il faut noter qu'on peut avoir $x = w$. Dans ce cas, on a nécessairement $a \neq p$ puisque (x, a) est une valeur courante tandis que (w, p) a été éliminée. Notons que AFVC peut être perçue comme une cohérence locale puisqu'il suffit de raisonner à partir de l'ensemble conflit de chaque valeur en échec. En particulier pour les réseaux binaires, une valeur (x, a) est AFVC-cohérente vis à vis d'une valeur en échec (w, p) ssi (x, a) est compatible avec une valeur valide dans $\chi(w, p)$. Un algorithme pour établir AFVC et basé sur cette simple observation est présenté plus loin.

Le résultat suivant est important :

Proposition 2. *Toute valeur AFVC-incohérente est globalement incohérente.*

Démonstration. Soit (x, a) une valeur courante de P qui est AFVC-incohérente vis à vis d'une valeur en échec (w, p) de P. Supposons qu'il existe une solution S de P telle que $S[x] = a$. Nécessairement, puisque (x, a) est AFVC-incohérente vis à vis de (w, p), $S[w/p]$ ne viole aucune contrainte portant sur w, et donc est une solution de P. Cela contredit le fait que (w, p) soit globalement incohérente (puisque c'est une valeur en échec), et donc notre hypothèse. On en déduit que (x, a) est globalement incohérente. □

La figure 3 présente un fragment d'un réseau de contraintes qui peut être complété pour faire en sorte que les cohérences usuelles (cohérence d'arc,...) soient vérifiées. On suppose que (w, p) est une valeur en échec et que $\chi(w, p) = \{\{(x, a)\}, \{(y, c)\}\}$. Il est facile de vérifier que (w, a) et (z, a) sont AFVC-incohérentes. En effet, (w, a) et (z, a) ne sont compatibles avec aucune valeur de $\chi(w, p)$.

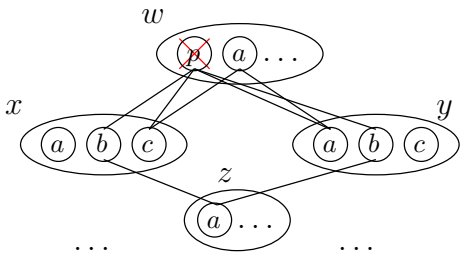


FIG. 3 – Illustration de AFVC

Quand on se restreint à des nogoods de taille 1, FVC est plus faible que AFVC (preuve omise). On a :

Proposition 3. *Soit (x, a) une valeur de P. Si $I = \{(x, a)\}$ est FVC-incohérente alors (x, a) est AFVC-incohérente.*

Enfin, on peut montrer que AFVC vérifie les propriétés (cf. [1, 4]) qui permettent de définir la clôture AFVC.

Proposition 4. *Pour tout réseau P, il existe un plus grand réseau AFVC-cohérent équivalent à P, appelé la clôture AFVC de P et noté $AFVC(P)$, tel que $AFVC(P) \preceq P$.*

[1] Une approche similaire bien que différente du point de vue opérationnel est de décomposer le problème [14]

$AFVC(P)$ peut s'obtenir en éliminant itérativement, dans un ordre quelconque, les valeurs qui ne sont pas AFVC-cohérentes.

4 Algorithme pour établir AFVC sur les réseaux binaires

À partir d'un réseau binaire P et d'un ensemble D de valeurs en échec, la procédure établirAFVC (algorithme 1) calcule $AFVC(P)$ ou lance une exception si un domaine devient vide. Les structures de données utilisées sont les suivantes. Pour chaque valeur en échec (w,p), $\chi(w,p)$ est un tableau de valeurs indexées de 1 à $length(\chi(w,p))$. Ces valeurs correspondent aux instanciations (de taille 1 puisque P est binaire) de l'ensemble conflit de (w,p). À chaque couple composé d'une valeur (x,a) de P et d'une valeur en échec (w,p) de D, le tableau à deux dimensions $last$ associe un entier qui est l'indice de la dernière valeur de $\chi(w,p)$ identifiée comme compatible avec (x,a) : $last[(x,a)][(w,p)]$ donne la position du dernier support AFVC trouvé pour (x,a) sur (w,p). Pour chaque valeur (y,b) de P, $S(y,b)$ est une liste stockant les couples (valeur,valeur en échec) pour lesquels (y,b) est le dernier support AFVC identifié. Les structures S et $last$ sont inspirées de celles de AC6 et AC2001 (voir [4]).

La procédure initialiser initialise les structures de données : les ensembles conflit $\chi(w,p)$ sont construits et les listes $S(x,a)$) sont vidées. La procédure établirAFVC tente d'identifier un support AFVC pour chaque couple (valeur,valeur en échec). Si aucun support n'est trouvé pour (x,a) sur une valeur en échec, a est retiré de $dom(x)$ et ajouté à la file de propagation Q. Chaque valeur $(y,b) \in Q$ est propagée : un nouveau support est recherché pour chaque couple de $S(y,b)$ (à partir de la dernière position enregistrée).

```
1  procedure initialiser(P : CN, D : ensemble de valeurs en échec)
2  begin
3      foreach valeur en échec (w,p) ∈ D do
4          χ(w,p) ← ∅
5          foreach voisin x de w do
6              foreach valeur a ∈ dom(x) do
7                  if (w,p) et (x,a) sont incompatibles then
8                      ajouter (x,a) à χ(w,p)
9          if χ(w,p) = ∅ then
10             throw INCOHÉRENCE
11     foreach valeur (x,a) de P do
12         S(x,a) ← ∅
13 end
```

```
1  procedure rechercheSupportFV((x,a) : valeur, (w,p) : valeur en échec)
2  begin
3      position ← last[(x,a)][(w,p)] + 1
4      while position ≤ length(χ(w,p)) do
5          (y,b) ← χ(w,p)[position]
6          if b ∈ dom(y) ∧ (x,a) et (y,b) sont compatibles then
7              last[(x,a)][(w,p)] ← position
8              ajouter ((x,a),(w,p)) à S(y,b)
9              return
10         position ← position + 1
11     retirer a de dom(x)
12     if dom(x) = ∅ then
13         throw INCOHÉRENCE
14     ajouter (x,a) à Q
15 end
16 procedure établirAFVC(P : CN, D : ensemble de valeurs en échec)
17 begin
18     Q ← ∅
19     foreach valeur (x,a) de P do
20         foreach valeur en échec (w,p) ∈ D do
21             last[(x,a)][(w,p)] ← 0
22             rechercheSupportFV((x,a),(w,p))
23     while Q ≠ ∅ do
24         extraire (y,b) de Q
25         foreach ((x,a),(w,p)) de S(y,b) do
26             rechercheSupportFV((x,a),(w,p))
27         S(y,b) ← ∅
28 end
```

Algorithm 1: Etablir AFVC

Dans le pire des cas, l'algorithme a une complexité en espace de $O(pM + pnd + pnd) = O(pnd)$ et une complexité en temps de $O(pMnd)$, avec n le nombre de variables, d la taille du plus grand domaine, p le nombre de valeurs en échec ($p = |D|$) et M la taille maximale d'un ensemble conflit ($M = max_{(w,p)\in D}|\chi(w,p)|$). Dans le détail, initialiser a une complexité en temps de $O(pnd)$, la complexité cumulée de rechercheSupportFV pour chaque couple (valeur,valeur en échec) est $O(M)$, et il y a $O(pnd)$ couple différent. Par définition, $p < nd$ et $M < nd$ donc $pMnd < n^3d^3$. En pratique, il est sans doute intéressant de borner par une constante le nombre de valeurs en échec et/ou la taille maximale des ensembles conflit pour se concentrer sur les valeurs en échec prometteuses. En bornant ces deux paramètres, la complexité devient $O(nd)$.

Ces complexités semblent satisfaisantes (par exemple en comparaison de $O(ed^2)$ pour un algorithme AC optimal en temps). Il est facile d'adapter cet algorithme pour l'utiliser

à chaque étape d'une recherche arborescente (la complexité reste la même sur une branche de l'arbre).

5 Substiuabilité et cohérences usuelles

La substituabilité de voisinage est une forme affaiblie de substituabilité [12] qui peut être rapprochée des cohérences basées sur les valeurs en échec. Une valeur $a \in dom(x)$ est voisin-substituable à une valeur $b \in dom(x)$ ssi pour chaque contrainte c portant sur x et tout support I pour (x, b) sur c, $I[x/a]$ est un support pour (x, a) sur c. La satisfiabilité des instances est préservée quand les valeurs voisin-substituables à une autre sont retirées. La sustituabilité de voisinage est utilisable pour réduire l'espace de recherche pour une instance tout en préservant la satisfiabilité [2]. Par exemple, elle peut être exploitée en tant qu'opérateur de réduction en appliquant une séquence convergente de substitutions de voisinage [7]. Cette notion est clairement liée au concept de symétrie [6]. La proposition suivante établit un lien avec AFVC.

Proposition 5. *Si une valeur (x, a) est voisin-substituable à une valeur (x, b) dans un réseau $P' \succ P$ et si (x, a) est une valeur en échec de P et (x, b) une valeur courante de P, alors (x, b) est AFVC-incohérente.*

Démonstration. La définition de substituabilité de voisinage peut être reformulée en : (x, a) est voisin-substituable à (x, b) ssi $\chi(x, a) \subseteq \chi(x, b)$. Si (x, a) est une valeur en échec de P, alors il n'est pas possible d'étendre (x, b) en une instanciation cohérente qui satisfasse (x, a). (x, b) est donc AFVC-incohérente. □

On peut voir AFVC comme un mécanisme paresseux et dynamique permettant d'identifier les valeurs substituables (et globalement incohérentes). Mais en fait, AFVC permet aussi d'identifier des valeurs incohérentes qui ne sont pas substituables. En fait, une valeur (x, b) est AFVC-incohérente si l'ensemble conflit de (x, b) est inclus dans l'ensemble conflit d'une quelconque valeur en échec. À la différence de la substituabilité de voisinage qui ne considère que les valeurs d'une même variable, AFVC peut identifier des valeurs dans des variables différentes et est donc plus générale de ce point de vue. Un exemple est donné en figure 3 : (w, p) est substituable à (w, a) mais pas à (z, a) puisque $w \neq z$.

Proposition 6. *AFVC et AC sont incomparables.*

Démonstration. Trivialement, AFVC $\not\rhd$ AC car il suffit de considérer un réseau qui n'est pas arc-cohérent et qui ne contient aucune valeur en échec. Par ailleurs, AC $\not\rhd$ AFVC. En effet, la figure 4 présente un réseau qui est arc-cohérent mais pas AFVC-cohérent. Plus précisément, le réseau P situé en haut de la figure est clairement arc-cohérent et AFVC-cohérent (puisqu'il n'y a pas de valeurs en échec). Supposons maintenant que la valeur (x, c) a été identifiée comme globalement incohérente (durant la recherche par exemple) : (x, c) est alors éliminé de $dom(x)$ pour aboutir au réseau P' situé en bas de la figure. Par définition, (x, c) est une valeur en échec, et on a $\chi(x, c) = \{\{(y, b)\}, \{(z, b)\}\}$. On peut observer que même si P' est arc-cohérent, P' n'est pas AFVC-cohérent car (x, a) n'est pas AFVC-cohérent. □

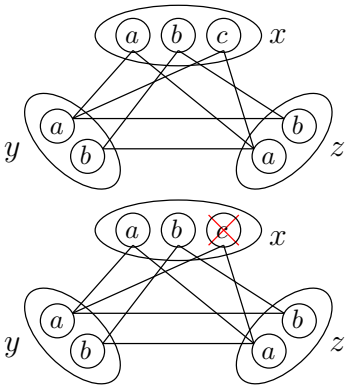


FIG. 4 – AC $\not\rhd$ AFVC

Il est intéressant de constater que AFVC est incomparable avec la plupart des cohérences réduisant les domaines. Plus précisément, AFVC est incomparable avec les cohérences «usuelles», i.e. les cohérences locales ϕ qui vérifient trois propriétés de base : a) ϕ est vérifiée sur tout réseau ne contenant que des contraintes universelles (une contrainte est universelle si elle est satisfaite par toute instanciation valide des variables dans sa portée), b) ϕ est vérifiée sur un réseau ssi elle est vérifiée sur chacune des composantes connexes du réseau, et c) il existe des réseaux insatisfiables pour lesquels ϕ est vérifiée. Par exemple, (G)AC, SAC, PIC, ... sont usuelles, mais la cohérence globale (définie comme toute instanciation localement cohérente peut être étendue en une solution) ne l'est pas.

Proposition 7. *AFVC est incomparable avec les cohérences usuelles.*

Démonstration. Soit ϕ une cohérence locale usuelle, un réseau P_1 qui ne contient que des contraintes universelles (donc satisfiable) et un réseau P_2 défini sur un jeu différent de variables qui soit insatisfiable mais néanmoins ϕ-cohérent. On considère le problème $P = P_1 \cup P_2$. Puisque P_2 est insatisfiable, toute valeur (x, a) de P_1 sera identifiée comme une valeur en échec lors de la recherche. Soit (x, a) l'une des valeurs en échec de P_1. Comme P_1 ne contient que des contraintes universelles, $\chi(x, a) = \varnothing$ et donc, aucune instanciation ne peut satisfaire la valeur en échec (x, a). De ce fait, P n'est pas AFVC-cohérent. En revanche, ϕ est vérifié sur P (conséquence des hypothèses), et donc $\phi \not\rhd$ AFVC. De l'autre côté, AFVC $\not\rhd \phi$: il suffit

de choisir un réseau P initial (donc sans valeurs en échec) et tel que ϕ ne soit pas vérifié. Par conséquent, AFVC et ϕ sont incomparables. □

6 Une hiérarchie de cohérences basées sur les valeurs en échec

Dans [11], Freuder a introduit la famille des (i, j)-cohérences. De manière informelle, un réseau de contraintes est (i, j)-cohérent ssi toute instanciation localement cohérente d'un ensemble de i variables peut être étendue en une instanciation localement cohérente contenant n'importe quel ensemble de j variables additionnelles. La cohérence d'arc, la cohérence de chemin [19, 18] et la cohérence de chemin inverse (PIC) [13] appartiennent toutes à cette famille puisqu'elles correspondent respectivement à la $(1, 1)$-cohérence, la $(2, 1)$-cohérence et la $(1, 2)$-cohérence. Une autre famille importante de cohérences basées cette fois-ci sur les contraintes est celle des (i, m)-cohérences relationnelles [9]. Informellement, un réseau est (i, m)-cohérent relationnel ssi pour chaque ensemble C de m contraintes avec $Y = \cup_{c \in C} scp(c)$ et tout ensemble $X \subseteq Y$ de i variables, toute instanciation localement cohérente de X peut être étendue en une instanciation valide de Y satisfaisant chaque contrainte de C. La cohérence d'arc généralisée et la cohérence de chemin inverse relationnelle [5] correspondent respectivement à la $(1, 1)$-cohérence relationnelle et la $(1, 2)$-cohérence relationnelle.

En s'inspirant de ces schémas, nous proposons une nouvelle famille de cohérences fondées sur la notion de valeurs en échec.

Définition 6 (FV-(i, p)-cohérence)**.** *P est FV-(i, p)-cohérent ssi pour chaque ensemble X de i variables de P et tout ensemble Y de p valeurs en échec de P, toute instanciation localement cohérente de X peut être étendue en une instanciation localement cohérente satisfaisant chaque valeur en échec de Y.*

De nombreuses cohérences découlent de cette définition générale : la FV-cohérence d'arc (AFVC) est la FV-$(1, 1)$-cohérence ; la FV-cohérence de chemin (PFVC) est la FV-$(2, 1)$-cohérence et la FV cohérence de chemin inverse (PIFVC) est la FV-$(1, 2)$-cohérence.

Par analogie avec les cohérences MaxRPC et SAC (voir par exemple [4]) basées sur les variables, nous définissons deux nouvelles cohérences.

Définition 7 (MaxFVC)**.** *Une valeur (x, a) de P est MaxFVC-cohérente ssi pour chaque valeur en échec (w, p) de P, (x, a) peut être étendue en une instanciation localement cohérente I satisfaisant (w, p) et telle que pour toute valeur en échec additionnelle (w', p') de P, I peut être étendue en une instanciation localement cohérente satisfaisant (w', p').*

Définition 8 (SAFVC)**.** *Une valeur (x, a) de P est SAFVC-cohérente ssi $AFVC(P|_{x=a}) \neq \bot$.*

Proposition 8. *PIFVC $\rhd$ AFVC et SAFVC $\rhd$ AFVC.*

Démonstration. Il découle des définitions que PIFVC $\trianglerighteq$ AFVC et SAFVC $\trianglerighteq$ AFVC. La figure 5 présente un exemple montrant l'ordre strict. Le réseau P a deux valeurs en échec (w, p), et (x, p). De plus, $\chi(w, p) = \{\{(y, b)\}, \{(z, b)\}\}$ et $\chi(x, p) = \{\{(y, a)\}, \{(z, b)\}\}$. P est AFVC-cohérent mais n'est ni SAFVC-cohérent, ni PIFVC-cohérent. En effet, (z, a) ne peut être étendu à une instanciation satisfaisant les deux valeurs en échec simultanément et $AFVC(P|_{z=a}) = \bot$. □

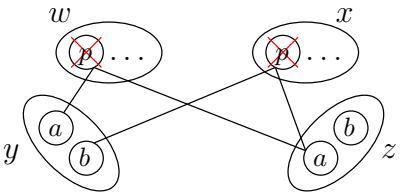


FIG. 5 – AFVC est vérifiée mais ni SAFVC, ni PIFVC ne le sont.

Proposition 9. *MaxFVC $\rhd$ PIFVC.*

Démonstration. Par définition, MaxFVC $\trianglerighteq$ PIFVC. La figure 6 permet de montrer l'ordre strict. Le réseau P possède trois valeurs en échec (w, p), (x, p) et (y, p). P est PIFVC-cohérent. En effet, chaque valeur de u (resp. v) est compatible avec (v, c) (resp. (u, c)) qui satisfait les trois valeurs en échec. Pour z, (z, b) est clairement PIFVC-cohérent tandis que pour (z, a), les valeurs en échec (w, p) et (x, p) sont satisfaites par $\{(z, a), (u, a), (v, a)\}$, les valeurs en échec (w, p) et (y, p) par $\{(z, a), (u, a), (v, b)\}$ et les valeurs en échec (x, p) et (y, p) par $\{(z, a), (u, b), (v, b)\}$. P n'est pas MaxFVC-cohérent. En effet, $\{(z, a), (u, b)\}$ et $\{(z, a), (v, a)\}$ sont les seules extensions de (z, a) qui satisfont (x, p). Cependant, pour $\{(z, a), (u, b)\}$, il est impossible de trouver une extension qui satisfasse (w, p) et pour $\{(z, a), (v, a)\}$, il est impossible de trouver une extension qui satisfasse (y, p). □

On peut aussi montrer que PIFVC et SAFVC d'une part, et MaxFVC et SAFVC d'autre part, sont incomparables.

Proposition 10. *PIFVC et SAFVC sont incomparables.*

Démonstration. Le réseau P de la figure 7 est SAFVC-cohérent mais pas PIFVC-cohérent. La valeur (z, a) ne peut être étendue pour satisfaire les deux valeurs en échec (x, p) et $y, p)$. Donc, le réseau est PIFVC-incohérent. De ce fait, SAFVC n'est pas plus fort que PIFVC. Par ailleurs,

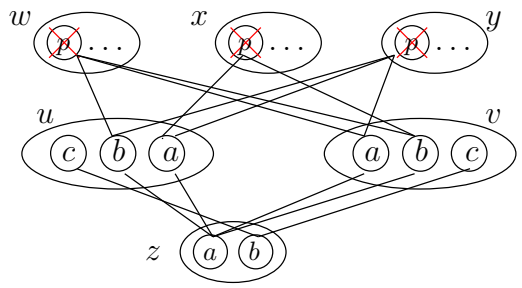


FIG. 6 – SAFVC et PIFVC sont vérifiées mais MaxFVC ne l'est pas.

la figure 6 représente un réseau qui n'est pas MaxFVC-cohérent mais qui est SAFVC-cohérent. Puisque MaxFVC est plus fort que PIFVC, on en déduit que PIFVC n'est pas plus fort que SAFVC. Donc, PIFVC et SAFVC sont incomparables. □

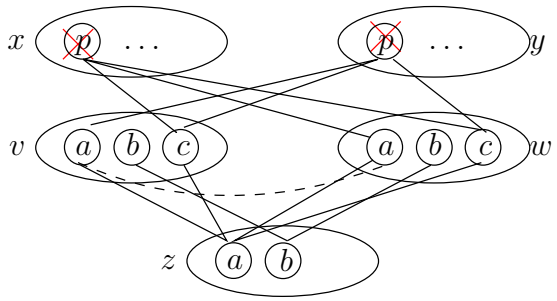


FIG. 7 – SAFVC est vérifié mais PIFVC ne l'est pas. Le seul tuple interdit entre v et w est (a, a).

Proposition 11. *MaxFVC et SAFVC sont incomparables.*

Démonstration. Le réseau de la figure 6 n'est pas MaxFVC-cohérent mais est SAFVC-cohérent. De ce fait, SAFVC n'est pas plus fort que MaxFVC. Considérons maintenant le réseau de la figure 7 auquel on apporte deux modifications : la valeur (z, b) est retirée du domaine initial de la variable et l'incompatibilité entre (v, a) et (w, a) est également effacée. Le réseau résultant est MaxFVC-cohérent mais pas SAFVC-cohérent. En effet, $AFVC(P|_{w=c}) = \bot$. □

La figure 8 résume les relations entre les différentes cohérences basées sur les valeurs en échec.

7 Conclusion

Cet article montre comment les valeurs détectées comme globalement incohérentes pendant la recherche et nommées valeurs en échec (FV) peuvent être utilisées pour élaguer l'arbre de recherche par l'intermédiaire d'une nouvelle famille de cohérences locales qui est orthogonale aux cohérences habituelles. Etablir AFVC (la FV-cohérence d'arc) peut être réalisé avec une complexité relativement faible et permet de détecter de manière paresseuse une forme généralisée de la substituabilié de voisinage.

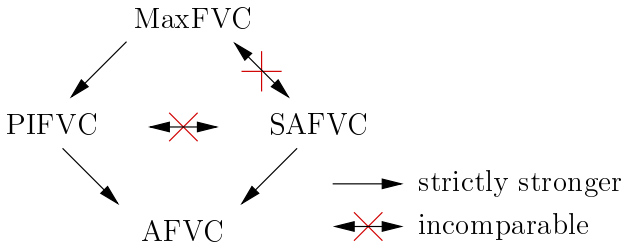


FIG. 8 – Relations entre les cohérences basées sur les valeurs en échec.

Bien qu'il soit cousin des approches qui éliminent les redondances en postant des contraintes [20, 15] ou en décomposant le problème [14], ce mode de raisonnement est différent car il définit une hiérarchie de cohérences de plus en plus fortes.

Pour les réseaux binaires, les valeurs en échec permettent d'identifier et d'exploiter une forme de nogood dans le même esprit que les nogoods généralisés de [16], les global cut seed de [10], les noyaux de [21] et les états partiels de [17]. Nous pensons que l'identification des propriétés communes de ces différentes approches est une perspective intéressante. Une autre perspective prometteuse est la mise au point d'algorithmes efficaces pour AFVC dans le cas n-aire.

Remerciements

Ce travail a bénéficié du soutien de l'IUT de Lens et du CNRS.

Références

[1] K.R. Apt. *Principles of Constraint Programming*. Cambridge University Press, 2003.

[2] A. Bellicha, C. Capelle, M. Habib, T. Kokény, and M.C. Vilarem. CSP techniques using partial orders on domain values. In *Proceedings of ECAI'94 workshop on constraint satisfaction issues raised by practical applications*, 1994.

[3] H. Bennaceur, C. Lecoutre, and O. Roussel. A decomposition technique for solving Max-CSP. In *Proceedings of ECAI'08*, pages 500–504, 2008.

[4] C. Bessiere. Constraint propagation. In *Handbook of Constraint Programming*, chapter 3. Elsevier, 2006.

[5] C. Bessiere, K. Stergiou, and T. Walsh. Domain filtering consistencies for non-binary constraints. *Artificial Intelligence*, 72(6-7) :800–822, 2008.

[6] D. Cohen, P. Jeavons, C. Jefferson, K.E. Petrie, and B. Smith. Symmetry definitions for constraint satisfaction problems. *Constraints*, 11(2-3) :115–137, 2006.

[7] M.C. Cooper. Fundamental properties of neighbourhood substitution in constraint satisfaction problems. *Artificial Intelligence*, 90 :1–24, 1997.

[8] R. Debruyne and C. Bessiere. Domain filtering consistencies. *Journal of Artificial Intelligence Research*, 14 :205–230, 2001.

[9] R. Dechter and P. van Beek. Local and global relational consistency. *Theoretical Computer Science*, 173(1) :283–308, 1997.

[10] F. Focacci and M. Milano. Global cut framework for removing symmetries. In *Proceedings of CP'01*, pages 77–92, 2001.

[11] E.C. Freuder. A sufficient condition for backtrack-bounded search. *Journal of the ACM*, 32(4) :755–761, 1985.

[12] E.C. Freuder. Eliminating interchangeable values in constraint satisfaction problems. In *Proceedings of AAAI'91*, pages 227–233, 1991.

[13] E.C. Freuder and C. Elfe. Neighborhood inverse consistency preprocessing. In *Proceedings of AAAI'96*, pages 202–208, 1996.

[14] E.C. Freuder and P.D. Hubbe. Using inferred disjunctive constraints to decompose constraint satisfaction problems. In *Proceedings of IJCAI'93*, pages 254–261, 1993.

[15] G. Gallo and G. Urbani. Algorithms for testing the satisfiability of propositional formulae. *Journal of Logic Programming*, 7(1) :45–61, 1989.

[16] G. Katsirelos and F. Bacchus. Generalized nogoods in CSPs. In *Proceedings of AAAI'05*, pages 390–396, 2005.

[17] C. Lecoutre, L. Sais, S. Tabary, and V. Vidal. Exploiting past and future : Pruning by inconsistent partial state dominance. In *Proceedings of CP'07*, pages 453–467, 2007.

[18] A.K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1) :99–118, 1977.

[19] U. Montanari. Network of constraints : Fundamental properties and applications to picture processing. *Information Science*, 7 :95–132, 1974.

[20] P.W. Purdom. Solving satisfiability with less searching. *IEEE transactions on pattern analysis and machine intelligence*, 6(4) :510–513, 1984.

[21] I. Razgon and A. Meisels. A CSP search algorithm with responsibility sets and kernels. *Constraints*, 12(2) :151–177, 2007.

Identification et exploitation d'états partiels

Christophe Lecoutre Sébastien Tabary Vincent Vidal

CRIL – CNRS UMR 8188,
Université Lille-Nord de France, Artois
rue de l'université, SP 16, F-62307 Lens
{lecoutre,tabary,vidal}@cril.fr

Résumé

Dans cet article, nous proposons une étude concernant l'identification et l'exploitation des états partiels inconsistants (IPS) dans le cadre de la résolution du problème de satisfaction de contraintes (CSP). Un IPS correspond à un ensemble de "décisions d'appartenance" prises sur un réseau de contraintes et ne menant à aucune solution. Notre étude tend à unifier un certain nombre de travaux récents concernant les nogoods généralisés, le concept de valeurs en échec et la détection de dominance. Nous introduisons un formalisme simple et unificateur et montrons comment la cohérence d'arc généralisée (GAC) peut être établie efficacement sur les contraintes associées aux IPSs tout au long de la recherche. Nous identifions également plusieurs opérateurs qui permettent d'extraire des IPSs à chaque noeud de l'arbre de recherche prouvé comme étant la racine d'un sous-arbre sans solutions.

1 Introduction

Ces dernières années, plusieurs formes d'apprentissage sophistiquées, basées essentiellement sur la détection de conflits, ont été proposées pour la résolution du problème de satisfaction de contraintes (CSP). Il s'agit de travaux concernant la détection de dominance (e.g. [1, 8] pour SBDS et [6, 5] pour SBDD), les nogoods généralisés [10], la notion de valeurs en échec [7, 16, 11] et les états partiels [13, 12]. Ces différentes approches possèdent un certain nombre de points communs qui n'ont pas été clairement établis jusqu'à présent.

Dans cet article[1], nous proposons un cadre unificateur capturant les notions évoquées ci-dessus. Ce cadre est celui revisité des états partiels (inconsistants). Il permet la représentation naturelle et l'exploitation précise et efficace des informations apprises au cours de la recherche. Plus précisément, ces informations sont représentées sous forme de contraintes dites de dominance. La cohérence d'arc (généralisée) est maintenue efficacement sur ces contraintes à l'aide de la structure des "watched literals".

Nous proposons également deux opérateurs d'extraction d'états partiels inconsistants (IPSs) appliqués à chaque noeud de l'arbre de recherche. Un IPS correspond à un ensemble de "décisions d'appartenance" prises sur un réseau de contraintes et ne menant à aucune solution. De manière intuitive, plus la taille d'un IPS est petite (en terme de nombre de décisions) plus sa capacité d'élagage est importante. Le premier opérateur proposé élimine toute décision portant sur une variable dont le domaine courant peut être déduit de l'état partiel. Le second opérateur élimine les décisions portant sur les variables qui ne sont pas impliquées dans la preuve d'incohérence du sous-arbre exploré à partir du noeud courant. Ces deux opérateurs peuvent être combinés.

2 Formalisme

Un réseau de contraintes (CN) P est composé d'un ensemble fini de n variables, noté $vars(P)$, et d'un ensemble fini de e contraintes, noté $cons(P)$. Chaque variable x possède un domaine associé, noté $dom(x)$, qui contient l'ensemble fini des valeurs qui peuvent être assignées à x. Chaque contrainte c implique un ensemble de variables, appelé la *portée* de c et noté $scp(c)$. Chaque contrainte est définie par une relation, notée $rel(c)$, qui contient l'ensemble des tuples autorisés pour les variables impliquées dans c. Une contrainte *binaire* implique exactement 2 variables et un CN binaire implique seulement des contraintes binaires. Une contrainte c de P est *universelle* (entailed) si et seulement si tous les tuples construits à partir des domaines des variables de $scp(C)$ sont autorisés par c.

[1] Une partie de cet article est fondée sur [12], non publié en français.

Dans ce papier, nous considérerons comme étant donnés un CN initial P^{init} et un CN courant P dérivé de P^{init} en réduisant potentiellement les domaines des variables. Le domaine initial d'une variable x est noté $dom^{init}(x)$ tandis que le domaine courant est noté $dom^P(x)$ ou plus simplement $dom(x)$. Pour chaque variable x, nous avons toujours $dom(x) \subseteq dom^{init}(x)$ et nous décrivons ce phénomène par $P \preceq P^{init}$. Une valeur (courante) de P est un couple (x, a) avec $x \in vars(P)$ et $a \in dom(x)$.

Une solution d'un CN correspond à l'assignation d'une valeur à chaque variable telle que toutes les contraintes soient satisfaites. Un CN P est dit être *satisfaisable* si et seulement si il admet au moins une solution. Si le domaine d'une variable de P est vide, P est trivialement insatisfaisable, et ceci est noté $P = \bot$. Le problème de satisfaction de contraintes (CSP) est la tâche NP-difficile de déterminer si un CN donné est satisfaisable ou pas. Une instance CSP est définie par un CN qui est résolu soit en trouvant une solution ou alors en prouvant son insatisfaisabilité. Pour résoudre une instance CSP, on peut utiliser un algorithme de recherche en profondeur d'abord équipé d'un mécanisme de retours-arrières. A chaque étape de la recherche, l'assignation d'une variable est effectuée suivie par une étape de filtrage appelée propagation de contraintes et basée sur des propriétés comme la cohérence d'arc généralisée (GAC). Pour une définition de GAC, appelée cohérence d'arc (AC) lorsque les contraintes sont binaires, voir par exemple [4].

3 Décisions

Les décisions représentent la notion élémentaire utilisée dans cet article.

Définition 1. *Une décision positive (resp. négative) δ est une restriction de la forme $x = a$ (resp. $x \neq a$), où x est une variable et $a \in dom^{init}(x)$. Une décision δ sur un CN P est une décision positive ou négative impliquant une valeur de P.*

En d'autres termes, une décision positive est une *assignation de variable* et une décision négative est une *réfutation de valeur*. Lorsque les décisions sont prises sur un CN P, nous obtenons une nouveau CN $P|_\Delta$ qui correspond au plus grand CN P' tel que $P' \preceq P$ et $P'|_\Delta = P'$. Pour tout ensemble de décisions Δ, $vars(\Delta)$ représente l'ensemble des variables apparaissant dans les décisions de Δ. Tous les ensembles de décisions ne sont pas forcément bien-formés ; ceux menant systématiquement à un échec ne sont pas pertinents. Un ensemble de décisions Δ est dit être *bien-formé* si et seulement si il existe au moins un CN P tel que $vars(P) = vars(\Delta)$ et $P|_\Delta \neq \bot$.

Un nogood *standard* de P est un ensemble Δ de décisions positives sur P tel que $P|_\Delta$ est insatisfaisable. En considérant à la fois des décisions positives et négatives comme dans [6, 10], on obtient une généralisation des nogoods standards, appelée *nogoods généralisés* :

Définition 2. *Un ensemble de décisions Δ sur un CN P est un nogood généralisé de P si et seulement si $P|_\Delta$ est insatisfaisable.*

Clairement, un nogood (standard) est généralisé mais l'opposé n'est pas nécessairement vrai. Un nogood généralisé peut représenter un nombre exponentiel de nogoods standards [10]. On peut utiliser la notion de décisions d'appartenance pour manipuler différentes classes d'équivalence de nogoods.

Définition 3. *Une décision d'appartenance δ est une restriction de la forme $x \in S_x$, où x est une variable et $\emptyset \subset S_x \subseteq dom^{init}(x)$; δ est stricte si et seulement si $S_x \subset dom^{init}(x)$. Une décision d'appartenance δ sur un CN P est une décision d'appartenance $x \in S_x$ tel que $x \in vars(P)$; δ est stricte sur P si et seulement si $S_x \subset dom^P(x)$ et est valide sur P si et seulement si $S_x \subseteq dom^P(x)$.*

Un ensemble de décisions d'appartenance Δ est bien-formé si et seulement si chaque variable apparaît au plus une fois dans Δ.

Proposition 1. *Pour tout ensemble Δ bien-formé de décisions (positive et/ou négative) sur un CN P, il existe un unique ensemble bien-formé Δ^m de décisions d'appartenances strictes sur P tel que $P|_\Delta = P|_{\Delta^m}$.*

La preuve est omise. Par exemple, si $dom^{init}(x) = dom^{init}(y) = dom^{init}(z) = \{a, b, c\}$ et $\Delta = \{x = a, y \neq b, y \neq c, z \neq b\}$, alors nous avons $\Delta^m = \{x \in \{a\}, y \in \{a\}, z \in \{a, c\}\}$.

4 États partiels inconsistants

Nous introduisons maintenant le concept d'état partiel.

Définition 4. *Un état partiel Δ est un ensemble de décisions d'appartenance ; Δ est strict si et seulement si chaque décision d'appartenance de Δ est stricte. Un état partiel Δ sur un CN P est un ensemble bien-formé de décisions d'appartenance valides sur P ; Δ est strict sur P si et seulement si chaque décision d'appartenance de Δ est stricte sur P.*

Un état partiel strict impose que chaque décision d'appartenance constitue une réelle restriction (cf. Définition 3). Nous notons $vars(\Delta)$ l'ensemble des variables apparaissant dans les décisions d'appartenance de Δ. Si Δ est un état partiel et si $(x \in S_x) \in \Delta$, nous notons S_x par $dom^\Delta(x)$. Il est important de noter que si P est un CN impliquant x et tel que $dom^\Delta(x) \not\subseteq dom^P(x)$, Δ ne peut pas être un état partiel sur P ; chaque décision d'appartenance d'un état partiel sur P doit être valide sur P.

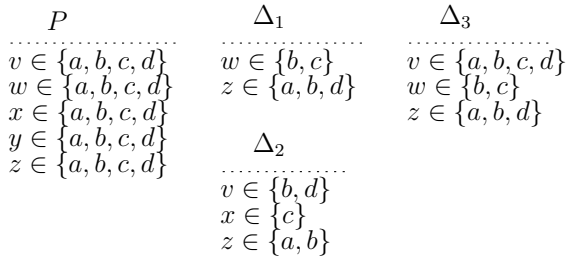


FIG. 1 – L'état courant d'un CN P et trois états partiels sur P. Contrairement à Δ_3, Δ_1 et Δ_2 sont stricts sur P.

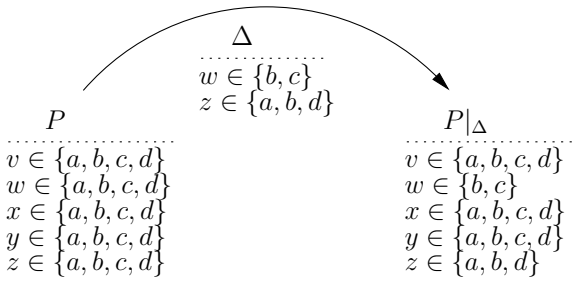


FIG. 2 – La restriction $P|_\Delta$ d'un CN P par un état partiel Δ sur P. Les états courants sont donnés pour P et $P|_\Delta$.

Il existe un état partiel immédiat pour tout CN P. Cet état partiel est construit en prenant en compte toutes les variables de P. Plus précisément, *l'état courant* d'un CN P est l'état partiel $\{(x \in dom^P(x)) \mid x \in vars(P)\}$ sur P. La Figure 1 présente l'état courant d'un CN P, et trois états partiels Δ_1, Δ_2 et Δ_3 sur P. Δ_1 et Δ_2 sont stricts sur P, mais Δ_3 n'est pas strict sur P car $dom^P(v) = dom^{\Delta_3}(v)$. Nous avons $vars(\Delta_1) = \{w, z\}$, $dom^{\Delta_1}(w) = \{b, c\}$ et $dom^{\Delta_1}(z) = \{a, b, d\}$.

La restriction $P|_\Delta$ d'un CN P par un état partiel Δ sur P est le CN obtenu à partir de P en restreignant le domaine de chaque variable $x \in vars(\Delta)$ à $dom^\Delta(x)$; pour chaque $x \in vars(\Delta)$, nous avons $dom^{P|_\Delta}(x) = dom^\Delta(x)$ et pour chaque $x \notin vars(\Delta)$, nous avons $dom^{P|_\Delta}(x) = dom^P(x)$.

Le réseau restreint $P|_\Delta$ est plus petit ($\preceq$) que P. Plus précisément, si $\Delta = \emptyset$, nous avons $P|_\Delta = P$ et si Δ est strict sur P et non vide, nous avons $P|_\Delta \prec P$. La figure 2 illustre la restriction d'un CN par un état partiel.

Quand une variable x n'apparaît pas dans un état partiel Δ, cela veut simplement dire que le domaine de x est considéré comme inchangé par Δ. En pratique, il est alors suffisant de ne manipuler que des états partiels stricts qui ont l'avantage d'être plus petits. Un état partiel strict peut être naturellement dérivé de n'importe quel état partiel.

Définition 5. *Soient P un CN et Δ un état partiel sur P. $\Delta^{s(P)}$ représente l'ensemble des décisions d'appartenance de Δ qui sont strictes sur P.*

Par exemple pour la figure 1, $\Delta_1 = \Delta_3^{s(P)}$. De manière importante, Δ et $\Delta^{s(P)}$ sont équivalents car $P|_\Delta = P|_{\Delta^{s(P)}}$. Un état partiel Δ sur un CN P est dit être inconsistant si le réseau défini comme la restriction de P sur Δ est insatisfaisable.

Définition 6. *Soient P un CN et Δ un état partiel sur P. Δ est un état partiel inconsistant (IPS) sur P, si et seulement si $P|_\Delta$ est insatisfaisable.*

Nous obtenons un IPS strict en ne tenant pas compte des décisions d'appartenance qui ne sont pas strictes.

Proposition 2. *Soient P un CN et Δ un IPS sur P. $\Delta^{s(P)}$ est un IPS sur P.*

La preuve est immédiate. La domination est définie comme suit.

Définition 7. *Soient P un CN et Δ un état partiel tel que $vars(\Delta) \subseteq vars(P)$. Δ domine P si et seulement si $\forall x \in vars(\Delta)$, $dom^P(x) \subseteq dom^\Delta(x)$.*

Notons que Δ n'est pas nécessairement un état partiel *sur* P. Lorsqu'un état partiel Δ domine un CN P, nous avons $P|_\Delta = P$. Par définition, un état partiel (non vide) strict Δ sur un CN P ne peut pas dominer P. Cette notion de dominance est utilisée pour les CNs strictement plus petit que P comme montré dans la figure 3.

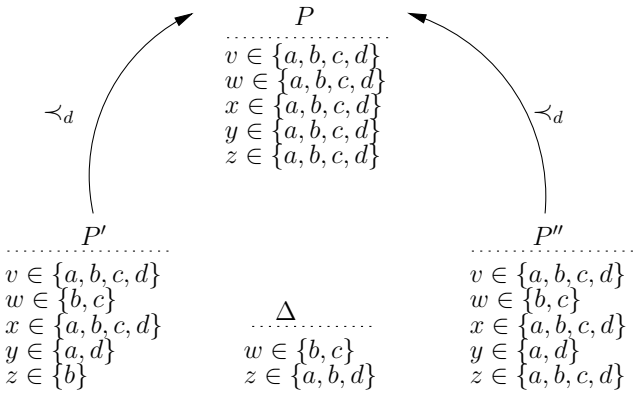


FIG. 3 – Deux CNs P' et P'' (strictement) plus petits que P et un état partiel Δ sur P. P' est dominé par Δ. P'' est presque dominé par Δ.

La proposition suivante est au coeur du raisonnement basé sur les états par détection de dominance.

Proposition 3. *Soient P et P' deux CNs tels que $P' \preceq P$, et Δ un IPS sur P. Si Δ domine P' alors P' est insatisfaisable.*

Démonstration. Si $P' \preceq P$, nous avons par monotonie $P'|_\Delta \preceq P|_\Delta$. Comme P' est dominé par Δ, $P'|_\Delta = P'$. On a alors $P' \preceq P|_\Delta$. Δ est un IPS sur P, ce qui veut dire que $P|_\Delta$ est insatisfaisable. Nous concluons que P' est insatisfaisable. □

Les états partiels sont dits être *globaux* lorsqu'ils sont définis sur le problème initial P^{init} ; dans le cas contraire, ils sont dits être *locaux*. Les IPSs globaux sont valides pour la totalité de l'espace de recherche tandis que les IPSs locaux ne sont utilisables que dans certains sous-arbres de recherche. Un IPS global peut toujours être extrait d'une impasse interne (noeud racine d'un sous-arbre sans solutions) de l'arbre de recherche construit par un algorithme de recherche avec retours-arrières : si v est une impasse interne et $P = cn(v)$ est le CN associé à v, l'état courant de P est un IPS lui-même. Cependant, cet IPS ne peut pas être exploité par la suite, excepté si des redémarrages sont effectuées ou si l'on exploite des symétries. Dans les sections suivantes, nous présentons plusieurs approches pour construire des IPSs pertinents.

5 Propager les IPSs

Dans le contexte d'un algorithme de recherche avec retours-arrières, la proposition 3 peut être exploitée pour élaguer des noeuds dominés par des IPSs précédemment identifiés. Plus précisément, à chaque fois que l'algorithme de recherche génère un nouveau noeud v, on peut vérifier si $cn(v)$, le CN associé à v, est dominé par un IPS préalablement enregistré. Deux mécanismes d'élagage peuvent être employés : soit v est directement rejeté parce qu'il est dominé, soit certains domaines des variables de $cn(v)$ sont filtrés pour garantir que la dominance ne peut apparaître plus tard. Pour clarifier cela, nous introduisons une version affaiblie de la dominance :

Définition 8. *Soient P un CN et Δ un état partiel tel que $vars(\Delta) \subseteq vars(P)$. Δ domine presque P si et seulement si il existe une décision d'appartenance $x \in S_x$ dans Δ telle que $dom^P(x) \not\subseteq dom^\Delta(x)$, $dom^P(x) \cap dom^\Delta(x) \neq \emptyset$ et $\Delta \setminus \{x \in S_x\}$ domine P.*

Lorsque la dominance ou la presque dominance implique un IPS, cela devient intéressant. Si un CN P est dominé par un IPS alors P est nécessairement insatisfaisable ; ceci est une conséquence de la proposition 3. D'un autre coté, un CN P qui est presque dominé par un IPS Δ peut être simplifié en supprimant toutes les valeurs qui pourraient rendre le CN dominé. En effet, si x est la seule variable de P telle que $dom^P(x) \not\subseteq dom^\Delta(x)$, nous pouvons déduire de manière certaine que chaque valeur de $dom^P(x) \cap dom^\Delta(x)$ est inconsistante. Ceci est appelé la cohérence 1-dominance dans [15]. Par exemple, le CN P'' de la figure 3 est presque dominé par Δ. Si Δ est un IPS sur P, les valeurs a, b et d du domaine de z peuvent être supprimées sans perte de solutions.

A strictement parler, comme montré ci-dessous, la cohérence 1-dominance n'est pas exactement une nouvelle cohérence si l'on considère que chaque IPS est représenté par une contrainte de *dominance*. Si Δ est un IPS, alors une contrainte de dominance c_Δ peut être construite telle que sa portée est $vars(\Delta)$ et sa relation interdit tous les tuples qui satisfont simultanément les décisions d'appartenance de Δ. Par exemple, soient x, y et z trois variables telles que $dom^{init}(x) = dom^{init}(y) = dom^{init}(z) = \{a, b, c\}$ et $\Delta = \{x \in \{a, b\}, y \in \{c\}, z \in \{b, c\}\}$ un IPS. La contrainte de dominance ternaire correspondant à Δ est c_Δ telle que $scp(c_\Delta) = \{x, y, z\}$ et $rel(c_\Delta) = dom^{init}(x) \times dom^{init}(y) \times dom^{init}(z) \setminus \{a, b\} \times \{c\} \times \{b, c\}$.

Les contraintes de dominance peuvent être exprimées en intension en appliquant simplement la loi de De Morgan sur les IPSs qui sont alors vus comme des conjonctions logiques de décisions d'appartenance. Par exemple, à partir de l'IPS $\Delta = \{x \in \{a, b\}, y \in \{c\}, z \in \{b, c\}\}$, on peut formuler la contrainte de dominance $c_\Delta : x \notin \{a, b\} \vee y \notin \{c\} \vee z \notin \{b, c\}$, ou de manière équivalente $c_\Delta : x \in \{c\} \vee y \in \{a, b\} \vee z \in \{a\}$ si $dom^{init}(x) = dom^{init}(y) = dom^{init}(z) = \{a, b, c\}$. De manière générale, pour chaque décision $x \in S_x$ apparaissant dans Δ, la décision complémentaire $x \in dom^{init}(x) \setminus S_x$ apparaît dans (l'expression du prédicat de) la contrainte de dominance c_Δ. On se référera à ces décisions complémentaires comme décisions de la contrainte c_Δ ; parfois, c_Δ est considéré comme un ensemble.

Les décisions d'appartenance peuvent être dans trois états différents selon le problème courant. Une décision d'appartenance $x \in S_x$ est dite *satisfaite* (i.e. toujours vraie) si et seulement si $dom(x) \subseteq S_x$. Une décision d'appartenance $x \in S_x$ est dite *falsifiée* (i.e. toujours fausse) si et seulement si $dom(x) \cap S_x = \emptyset$. Une décision d'appartenance qui est ni satisfaite ni falsifiée est dite *libre* (indéterminée) ; nous avons $\emptyset \subset dom(x) \setminus S_x \subset dom(x)$. Par exemple, si $dom(x) = \{a, b\}$, $dom(y) = \{b\}$ et $dom(z) = \{a, c\}$, alors $x \in \{c\} \vee y \in \{a, b\} \vee z \in \{a\}$ est une contrainte telle que la première décision est falsifiée, la seconde est satisfaite et la troisième est libre.

Un IPS Δ, vu comme une contrainte de dominance c_Δ, indique simplement qu'au moins une décision apparaissant dans l'expression du prédicat de c_Δ doit être évaluée à vrai. Quatre cas sont possibles lorsque l'on utilise une contrainte de dominance c_Δ :

1. c_Δ est universelle car une décision de c_Δ est satisfaite : le problème courant ne peut être dominé par Δ.
2. c_Δ est violée (disentailed) parce que toutes les décisions de c_Δ sont falsifiées : le problème courant est dominé par Δ (on doit alors effectuer un retour-arrière).
3. c_Δ ne contient aucune décision satisfaite et exactement une décision libre : le problème courant est presque dominé par Δ (cette décision libre peut être forcée afin de la satisfaire).
4. c_Δ ne contient aucune décision satisfaite et (au moins) deux décisions libres : le problème courant n'est ni dominé ni presque dominé par Δ.

On peut aisément observer que tant qu'il y a (au moins) deux décisions libres dans c_Δ, la contrainte c_Δ est GAC-cohérente. En d'autres termes, la contrainte c_Δ n'est plus GAC-cohérente lorsque Δ domine ou presque domine le réseau courant. Si le problème courant est dominé par un IPS Δ, cela veut dire que la contrainte c_Δ est violée et que par conséquent il faut effectuer un retour-arrière (si possible). Si le problème courant est presque dominé par un IPS Δ, le fait de forcer l'unique décision libre (à être satisfaite) revient à établir GAC en supprimant certaines valeurs du domaine de la variable impliquée dans cette décision. Par conséquent, contrôler la dominance et établir la cohérence 1-dominance [15] sur un IPS Δ est équivalent à établir GAC sur c_Δ.

La structure de données paresseuse des watched literals [20] peut être exploitée pour établir efficacement GAC sur des contraintes de dominance. Le principe est de marquer deux valeurs dans deux décisions d'appartenance **distinctes** de chaque contrainte de dominance c_Δ : ils permettent d'identifier le moment où un IPS domine ou presque domine le problème courant.

Par exemple, supposons que l'on ait identifié deux IPSs (sur P^{init}) $\Delta_1 = \{x \in \{c\}, y \in \{a\}, z \in \{a, b\}\}$ et $\Delta_2 = \{x \in \{b, c\}, w \in \{a, c\}\}$, et que l'on ait enregistré les contraintes de dominance $c_{\Delta_1} : x \in \{a, b\} \vee y \in \{b, c\} \vee z \in \{c\}$ et $c_{\Delta_2} : x \in \{a\} \vee w \in \{b\}$; le domaine initial de chaque variable est $\{a, b, c\}$. La figure 4 montre la base de contraintes de dominance, avec les valeurs (x, a) et (z, c) marquées dans la première contrainte de dominance et les valeurs (x, a) et (w, b) marquées dans la seconde. Les valeurs marquées sont indiquées par les marqueurs w_1 et w_2. Nous avons également un tableau de $O(nd)$ entrées (d est la taille du plus grand domaine) telles que chaque entrée correspond à une valeur (x, a) du problème (initial) et représente la tête d'une liste chaînée, notée $\mathscr{B}_{(x,a)}$, permettant l'accès aux contraintes de dominance de $\mathscr{B}$ qui contiennent (x, a) comme valeur marquée.

Pour chaque contrainte de dominance, tant que les deux valeurs marquées sont présentes, cela veut dire que les décisions d'appartenance dans lesquelles elles apparaissent sont libres (ou satisfaites) et que par conséquence la contrainte est GAC-cohérente. La propagation est guidée par les valeurs supprimées. Lorsqu'une valeur (x, a) est supprimée, la liste $\mathscr{B}_{(x,a)}$ est visitée. Pour chaque contrainte de dominance c de cette liste, une valeur *marquable*, i.e. une valeur présente dans le problème courant, doit être recherchée (mais pas dans le domaine contenant la seconde valeur marquée). Soit cette recherche aboutit et la valeur trouvée devient la nouvelle valeur marquée (remplaçant (x, a)), soit il faut forcer la décision impliquant la seconde valeur marquée de c_Δ afin de rendre c_Δ GAC-cohérente (potentiellement en générant un domaine vide).

Par exemple, la base de contraintes de dominance de la figure 5 est obtenue à partir de la situation illustrée par la figure 4 lorsque la valeur (z, c) est supprimée. Par conséquent, la valeur (y, b) est maintenant marquée au lieu de (z, c) dans c_{Δ_1}. Imaginons qu'un peu plus tard, l'on assigne la valeur c à la variable x ; les valeurs (x, a) et (x, b) sont alors supprimées. Les deux contraintes de dominance de $\mathscr{B}_{(x,a)}$ permettent d'effectuer des inférences (parce qu'il n'y a plus d'autres valeurs marquables). Ceci est montré par la figure 6.

Exploiter la technique des valeurs marquées sur des contraintes de dominance possède deux principaux avantages. Tout d'abord, la complexité dans le pire des cas pour établir GAC sur une contrainte de dominance c est linéaire en la taille de c. Plus précisément, si k est le nombre total de valeurs apparaissant dans les décisions d'appartenance de c, i.e. $k = \sum_{(x \in S_x) \in c} |S_x|$, alors c peut être rendu GAC-cohérent en $O(k)$; ce résultat est borné par $O(nd)$. Il suffit de contrôler si une valeur est marquable seulement une fois. Par exemple, w_1 peut parcourir la liste des valeurs apparaissant dans c de la gauche vers la droite tandis que w_2 peut parcourir la même liste dans l'ordre inverse. La recherche d'une valeur marquable est stoppée lorsque w_1 et w_2 font référence à la même décision d'appartenance. Deuxièmement, aucun travail de restauration n'est requis lorsque l'on effectue un retour-arrière. Non seulement cela permet d'économiser du temps, mais en plus cela facilite l'intégration de cette technique dans les solveurs de contraintes. Lorsqu'une valeur (x, a) est supprimée, la base de contraintes de dominance $\mathscr{B}$ est consultée, et plus précisément, les contraintes de dominance accessibles par $\mathscr{B}_{(x,a)}$ sont contrôlées. Quand des valeurs sont restaurées, aucune opération de maintenance n'est nécessaire.

Dans [9, 17], les auteurs ont suggéré l'utilisation de la structure des watched literals pour propager des nogoods généralisés. Comme mentionné dans la section précédente, les nogoods généralisés sont basiquement équivalents aux IPSs. Classiquement, les inférences sont effectuées par la propagation unitaire. Ce mécanisme est strictement plus faible que GAC. Par exemple, soit $x = a \vee x = b \vee y \neq c$ un nogood généralisé (exprimé ici comme une contrainte) tel que $dom^{init}(x) = dom^{init}(y) = \{a, b, c\}$, $dom(x) = \{a, b, c\}$ et $dom(y) = \{c\}$. Dans ce nogood, les deux décisions $x = a$ et $x = b$ sont libres, et donc aucune inférence ne peut être effectuée. L'IPS équivalent à ce nogood généralisé est $x \in \{a, b\} \vee y \in \{a, b\}$. En établissant GAC, on peut déduire $x \neq c$. Dans [17], une approche différente pour stocker et établir GAC sur des nogoods (généralisés) est également proposée. Les nogoods sont capturés par un automate qui permet une représentation compacte mais la compilation dynamique de ces nogoods semble difficile à mettre en place en pratique.

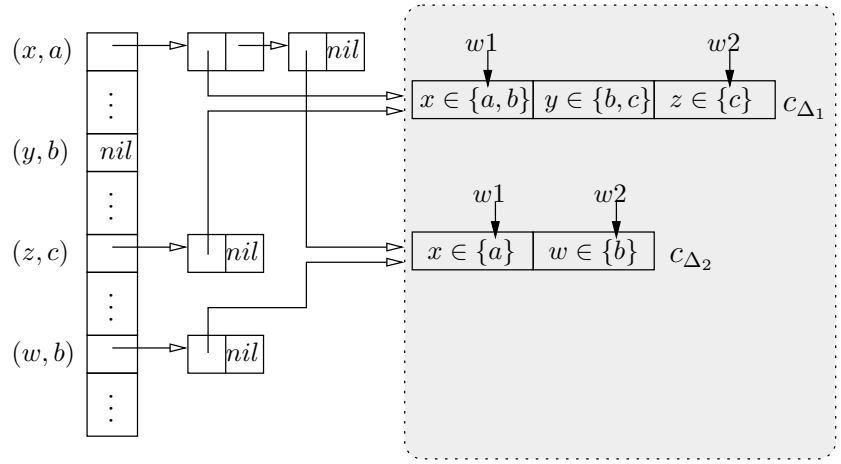


FIG. 4 – Une base de contraintes de dominance $\mathscr{B}$ incluant deux contraintes de dominance $c_{\Delta_1} : x \in \{a, b\} \vee y \in \{b, c\} \vee z \in \{c\}$ et $c_{\Delta_2} : x \in \{a\} \vee w \in \{b\}$.

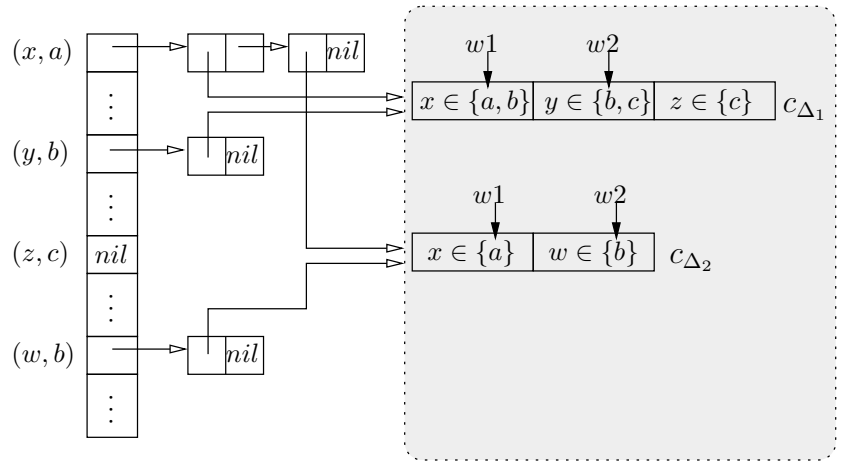


FIG. 5 – La base $\mathscr{B}$ de la figure 4 après que la valeur (z, c) soit éliminée. (y, b) est la nouvelle valeur marquée.

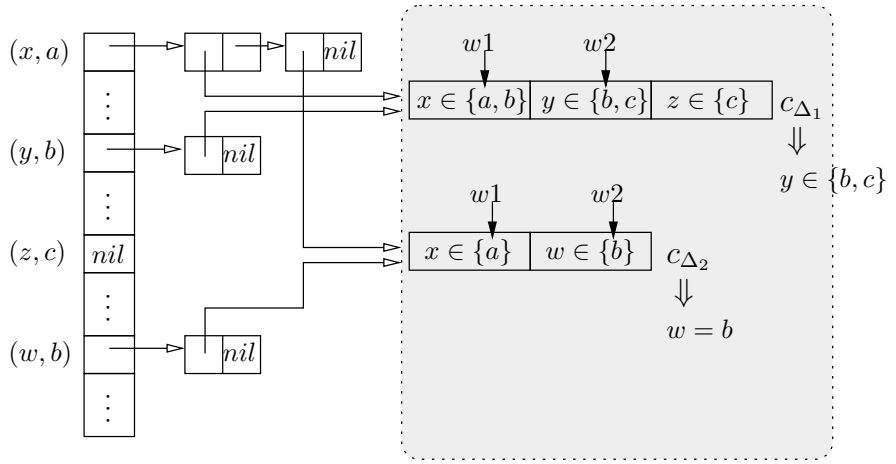


FIG. 6 – La base $\mathscr{B}$ de la figure 5 après que x soit assignée à la valeur c. Les décisions $y \in \{b, c\}$ et $w = b$ sont inferrées.

6 Identifier les IPSs

Les nogoods généralisés et la cohérence (FVC) basée sur les valeurs en échec [11] entrent dans le cadre des états partiels. Par manque de place, nous supprimons cette partie qui n'est pas nécessaire à la compréhension de l'article.

7 Opérateurs de réduction

Pour identifier des IPSs, on peut directement travailler avec les états issus d'impasses internes rencontrées au cours de la recherche. Les impasses internes sont des noeuds qui sont les racines de sous-arbres infructueux. Comme nous l'avons mentionné précédemment, l'état courant de chaque impasse interne est un IPS global. Plus précisément, si v est une impasse interne et $P = cn(v)$ le CN associé à v, alors l'état courant de P est un IPS sur P^{init}. De tels IPSs construits à partir d'impasses internes sont dits *élémentaires*.

Dans cette section, nous présentons plusieurs opérateurs dont le rôle est d'éliminer des variables (à strictement parler, des décisions d'appartenance) des IPSs élémentaires. Les états partiels obtenus après réduction sont des sous-ensembles d'états élémentaires dits *simples*.

Définition 9. *Un état partiel Δ sur un CN P est* simple *ssi* $\forall x \in vars(\Delta)$, $dom^{\Delta}(x) = dom^{P}(x)$.

Par exemple, considérons un CN P tel que $vars(P) = \{x, y, z\}$ avec $dom^P(x) = dom^P(y) = dom^P(z) = \{a, b, c\}$. $\Delta = \{x \in \{a, b, c\}, z \in \{a, b, c\}\}$ est un état partiel simple sur P tandis que $\Delta' = \{x \in \{a, b, c\}, z \in \{a, b\}\}$ est un état partiel sur P qui n'est pas simple car $dom^{\Delta'}(z) \neq dom^P(z)$.

Les états partiels simples obtenus après réduction (tel que proposé dans cette section) sont des IPSs globaux qui peuvent être exploités au cours de la recherche. Intuitivement, plus le nombre de variables impliquées dans un IPS est faible, plus grande sera sa capacité d'élagage. Cela contribue également à réduire la consommation mémoire.

Par la suite, on considère l'algorithme de recherche classique MAC (Maintaining Generalized Arc Consistency). MAC est un algorithme de recherche avec retours-arrières qui est basé sur un schéma de branchement binaire et qui maintient GAC pendant la recherche. Par conséquent, les inférences sont effectuées localement, i.e. au niveau d'une seule contrainte, pendant le processus de propagation de contraintes. GAC peut être établi en utilisant une collection de propagateurs associés à chaque contrainte. Ces propagateurs peuvent correspondre soit à une procédure générique de filtrage soit à une procédure de filtrage spécialisée (e.g. pour des contraintes globales).

7.1 Variables e-eliminables

Nous présentons un premier opérateur qui supprime les variables *e-eliminables*. Une variable e-eliminable est seulement impliquée dans des contraintes universelles ; ainsi elle ne peut plus jouer aucun rôle. ρ^{ent} est un opérateur qui élimine les variables e-eliminables et retourne un état partiel simple.

Définition 10. *Pour tout CN P, $\rho^{ent}(P)$ représente l'état partiel $\{(x \in dom^P(x)) \mid x \in vars(P) \wedge x$ n'est pas une variable e-eliminable de $P\}$.*

La proposition suivante établit le fait que ρ^{ent} est un opérateur permettant l'extraction d'un IPS à partir d'un CN insatisfaisable.

Proposition 4. *Si P est un CN insatisfaisable alors $\rho^{ent}(P)$ est un IPS sur tout CN $P' \succeq P$.*

De manière intéressante, cela veut dire que pour chaque impasse interne v rencontrée pendant la recherche, ρ^{ent} extrait un IPS sur P^{init} à partir de $cn(v)$, qui est un IPS global. Une illustration est donnée dans [13] (où ρ^{ent} est appelé ρ^{uni}).

7.2 Utilisation de preuves d'insatisfaisabilité

Proposition 5. *Si C est un noyau insatisfaisable d'un CN P alors $\Delta = \{(x \in dom^P(x)) \mid x \in vars(C)\}$ est un IPS sur tout CN $P' \succeq P$.*

En particulier, nous savons que Δ est un IPS global, i.e. un IPS sur P^{init}. De manière intéressante, il est possible d'identifier efficacement des noyaux insatisfaisables à chaque impasse interne en gardant la trace de toutes les contraintes impliquées dans des preuves d'insatisfaisabilité [2]. Ces contraintes sont celles utilisées durant la recherche pour supprimer, grâce à leurs propagateurs, au moins une valeur dans le domaine d'une variable. Cette approche peut être adaptée pour extraire des noyaux insatisfaisables à partir d'impasses internes en collectant des informations pertinentes dans les sous-arbres infructueux explorés.

L'algorithme 1 montre comment intégrer cette méthode à MAC. La fonction récursive MAC^{prf} détermine la satisfaisabilité d'un réseau P et retourne un couple composé d'une valeur booléenne (qui indique si P est satisfaisable ou pas), et d'un ensemble de variables. Cet ensemble est soit vide (quand P est satisfaisable) soit représente une preuve d'insatisfaisabilité. Une preuve est composée des variables impliquées dans la portée de contraintes qui ont déclenché au moins la suppression d'une valeur pendant la propagation.

A chaque noeud, une preuve est construite à partir de toutes les inférences produites pendant l'établissement de GAC, noté $GAC(P)$ à la ligne 2, et les preuves (lignes 6 et 8) associées aux sous-arbres gauche et droit (une fois qu'un

couple (x, a) a été sélectionné). Quand un noeud est prouvé être une impasse interne après avoir pris en considération les deux branches (la première étiquetée par $x = a$ et la seconde par $x \neq a$), une preuve d'insatisfaisabilité (entre les lignes 9 et 10) est obtenue en fusionnant simplement les preuves associées aux branches droite et gauche ; ici c'est pour P'. Notons que la complexité spatiale dans le pire des cas pour gérer les différentes preuves locales de l'arbre de recherche est $O(n^2d)$ puisqu'enregistrer une preuve est $O(n)$ et qu'il y a au plus $O(nd)$ noeuds par branche.

Algorithm 1: $\mathsf{MAC}^{prf}(P : \text{CN})$: (booléen, ensemble de variables)

1 $localProof \leftarrow \emptyset$
2 $P' \leftarrow GAC(P)$; // $localProof$ est maj par GAC
3 **if** $P' = \bot$ **then return** *(false,localProof)*
4 **if** $\forall x \in vars(P'), |dom(x)| = 1$ **then return** *(true,*$\emptyset$*)*
5 sélectionner une valeur (x, a) de P' telle que $|dom(x)| > 1$
6 $(sat, leftProof) \leftarrow \mathsf{MAC}^{prf}(P'|_{X=a})$
7 **if** sat **then return** *(true,*$\emptyset$*)*
8 $(sat, rightProof) \leftarrow \mathsf{MAC}^{prf}(P'|_{X\neq a})$
9 **if** sat **then return** *(true,*$\emptyset$*)*
// $leftProof \cup rightProof$ est une preuve pour P'
10 **return** $(false, localProof \cup leftProof \cup rightProof)$

En utilisant l'algorithme 1, on peut introduire un second opérateur qui ne sélectionne que les variables impliquées dans une preuve d'insatisfaisabilité. Cet opérateur peut être utilisé incrémentalement à n'importe quelle impasse interne d'un arbre construit par MAC.

Définition 11. *Soit P un CN tel que $\mathsf{MAC}^{prf}(P) = (false, proof)$. $\rho^{prf}(P)$ représente l'état partiel simple $\{(x \in dom^P(x)) \mid x \in proof\}$.*

La proposition suivante établit le fait que ρ^{prf} est un opérateur qui permet d'extraire des IPSs.

Proposition 6. *Si P est un CN insatisfaisable alors $\rho^{prf}(P)$ est un IPS sur tout CN $P' \succeq P$.*

Démonstration. Soit $\mathsf{MAC}^{prf}(P) = (false, proof)$. On peut montrer que $C = (proof, \{c \in cons(P) \mid scp(c) \subseteq proof\})$ est un noyau insatisfaisable de P. Nous déduisons le résultat de la définition de l'opérateur ρ^{prf} et de la proposition 5. □

Similairement à ρ^{ent}, ρ^{prf} permet d'extraire des IPSs globaux. En pratique, dans l'algorithme 1, l'opérateur ρ^{prf} peut être appelé pour extraire un IPS entre les lignes 9 et 10. La proposition suivante établit le fait que ρ^{prf} est plus performant que ρ^{ent} (i.e. permet d'extraire des IPSs représentant une plus large portion de l'espace de recherche).

Proposition 7. *Si P est un CN insatisfaisable alors $\rho^{prf}(P) \subseteq \rho^{ent}(P)$.*

Démonstration. Une contrainte universelle ne peut intervenir dans une preuve d'insatisfaisabilité. Une variable e-eliminable n'apparaît que dans des contraintes universelles, donc est nécessairement éliminée par ρ^{prf} (en considérant les hypothèses faites au début de la section). □

7.3 Utilisation de justifications

Le principe de l'opérateur présenté dans cette section est de construire un état partiel simple en éliminant les variables dont le domaine courant peut être déduit des autres. Ceci est possible en gardant la trace des contraintes à l'origine des suppressions de valeurs. Lorsqu'un propagateur associé à une contrainte c supprime une valeur (x, a), la contrainte c est enregistrée comme la *justification* de la suppression de (x, a). C'est une forme d'explication (de valeur éliminée) même si les explications sont classiquement basées sur des décisions (i.e. formées de décisions positives et négatives). Dans une certaine mesure, cela correspond à une utilisation basique de la définition générale de nogood proposée dans [18] qui inclut un ensemble de contraintes jouant le rôle de justification pour le nogood. Des justifications similaires ont été également exploitées pour établir AC sur des instances dynamiques [3].

Pour notre propos, nous avons besoin de raisonner avec un graphe d'implication à gros grain, appelé graphe de dépendance ici, et construit à partir des justifications. Lorsqu'une décision positive $x = a$ est prise (par l'algorithme de recherche), $just(x \neq b)$ est initialisé à nil pour toute valeur $b \in dom(x) \mid b \neq a$, et quand une décision négative $x \neq a$ est prise, $just(x \neq a)$ est également initialisé à nil. D'un autre coté, quand une valeur (x, a) est supprimée par un propagateur associé à une contrainte c, la justification de $x \neq a$ est simplement donnée par c : nous avons $just(x \neq a) = c$. Comme notre but est de circonscrire un état partiel simple, nous avons seulement besoin de savoir pour chaque valeur (x, a) supprimée, les variables responsables de sa suppression ; ce sont celles impliquées dans $just(x \neq a)$. A partir de ces informations, on peut construire un graphe orienté G ou les noeuds correspondent aux variables et les arcs aux dépendances entre les variables. Plus précisément, un arc existe dans G d'une variable x vers une variable y s'il existe une valeur supprimée (y, b) telle que sa justification soit une contrainte impliquant x. On ajoute également un noeud spécial nil, et un arc existe entre nil et une variable x si cette variable est impliquée dans une décision (positive ou négative) prise par l'algorithme de recherche, i.e. s'il existe une valeur supprimée (x, a) telle que sa justification soit nil. Le graphe de dépendance peut être utilisé pour réduire les IPSs ; ceci est décrit ci-dessous.

La figure 7 illustre notre propos. Sur la gauche de la figure, on représente le CN binaire initial P^{init}. P^{init} implique quatre variables et trois contraintes ; $vars(P^{init}) =$

$\{w, x, y, z\}$ et $cons(P^{init}) = \{w \neq x, x \geq y, x \leq z\}$. Le CN courant P est également représenté, il est obtenu à partir de P^{init} après avoir assigné la valeur 3 à w et établi AC. La figure 7 fournit les justifications des valeurs supprimées dans P ainsi que le graphe de dépendance construit à partir de ces explications. Les justifications sont obtenues comme suit. Lorsque la décision positive $w = 3$ est prise, les justifications de $w \neq 1$ et $w \neq 2$ sont initialisées à nil. Ces suppressions sont propagées à x grâce à la contrainte $w \neq x$, menant à la suppression de la valeur 3 de $dom(x)$; on obtient alors $just(x \neq 3) = (w \neq x)$. Cette nouvelle suppression est alors propagée aux variables y et z : la valeur 3 est supprimée de $dom(y)$ via la propagation de $x \geq y$ ce qui constitue sa justification et la valeur 0 est supprimée de $dom(z)$ via la propagation de $x \leq z$. Le graphe de dépendance est directement construit à partir de ces justifications.

Définition 12. *Soit x une variable de P et $a \in dom^{init}(x) \setminus dom^P(x)$. La justification de la suppression de (x, a), notée $just(x \neq a)$ est, si elle existe, la contrainte dont le propagateur associé a supprimé (x, a) sur le chemin menant de la racine de l'arbre de recherche au noeud v où $cn(v) = P$; sinon $just(x \neq a)$ est nil.*

Les justifications sont utilisées pour extraire un état partiel simple à partir d'un ensemble de variables X. Cet état partiel contient les variables de X qui ne peuvent pas être "expliquées" par X.

Définition 13. *Soit $X \subseteq vars(P)$ un ensemble de variables de P. Une variable $x \in X$ est j-eliminable de P vis-à-vis de X si et seulement si $\forall a \in dom^{P^{init}}(x) \setminus dom^P(x)$, $just(x \neq a)$ est une contrainte c telle que $c \neq nil$ et $scp(c) \subseteq X$.*

Le graphe de dépendance mentionné plus haut permet d'identifier directement ces variables. ρ^{jst} est un opérateur qui élimine les variables j-eliminables et retourne un état partiel simple.

Définition 14. *Soit $X \subseteq vars(P)$. $\rho^{jst}_X(P)$ est l'état partiel simple $\{(x \in dom^P(x)) \mid x \in X \wedge x$ n'est pas une variable j-eliminable de P vis à vis de $X\}$.*

Réduire des états partiels en éliminant les variables j-eliminables ne provoque pas fondamentalement de perte d'informations lorsqu'on considère P^{init} et GAC (la preuve est omise).

Proposition 8. *Soit Δ un état partiel simple sur P et $\Delta' = \rho^{jst}_{vars(\Delta)}(P)$. Nous avons : $GAC(P^{init}|_{\Delta'}) = GAC(P^{init}|_{\Delta})$.*

En utilisant la proposition 8, on peut montrer que pour tout CN P, $\rho^{jst}_{vars(P)}(P)$ produit un IPS global. Cependant, cet IPS n'est pas intéressant parce qu'il est (pour l'essentiel) équivalent à l'état courant P. En effet, c'est un état partiel avec toutes les variables impliquées dans une décision prise par l'algorithme de recherche. Cet IPS est équivalent au nogood généralisé correspondant à l'ensemble des décisions prises sur la branche menant de la racine de l'arbre de recherche à P. Heureusement, on peut utiliser l'opérateur ρ^{jst} après application d'un autre opérateur produisant un IPS (simple), comme montré dans le corollaire suivant.

Corollaire 1. *Soit Δ un état partiel simple sur P et $\Delta' = \rho^{jst}_{vars(\Delta)}(P)$. Si Δ est un IPS sur P^{init} alors Δ' est un IPS sur P^{init}.*

Démonstration. Si Δ est un IPS sur P^{init} alors $P^{init}|_{\Delta}$ est insatisfaisable. Comme GAC préserve la satisfaisabilité, et $GAC(P^{init}|_{\Delta'}) = GAC(P^{init}|_{\Delta})$ par la proposition 8, on peut déduire que $P^{init}|_{\Delta'}$ est insatisfaisable. Par conséquence Δ' est un IPS sur P^{init}. □

En conséquence du corollaire 1, on a la garantie que les deux opérateurs suivants produisent des IPSs à partir de CNs insatisfaisables.

Définition 15. *Soit P un CN.*
- $\rho^{jst \odot ent}(P) = \rho^{jst}_{vars(\Delta)}(P)$ *avec* $\Delta = \rho^{ent}(P)$.
- $\rho^{jst \odot prf}(P) = \rho^{jst}_{vars(\Delta)}(P)$ *avec* $\Delta = \rho^{prf}(P)$.

La figure 8 illustre (ici, sur des états partiels consistants) le comportement de ρ^{ent}, ρ^{jst} et leur combinaison $\rho^{jst \odot ent}$. Appliquer ρ^{ent} sur le réseau P de la figure 7 provoque l'élimination de w, menant à l'état partiel Δ_1, parce que w est seulement impliquée dans des contraintes universelles. En effet, la valeur restante 3 dans $dom(w)$ est compatible avec les deux valeurs restantes 1 et 2 de $dom(x)$ via la contrainte $w \neq x$. Les trois autres variables sont impliquées dans des contraintes qui ne sont pas universelles. Appliquer ρ^{jst} sur le réseau P vis-à-vis de $X = vars(P)$ provoque l'élimination de x, y et z menant à l'état partiel $\Delta_2 = \{w \in \{3\}\}$. En effet, w est la seule variable pour laquelle une suppression est justifiée par nil ; X étant $vars(P)$, c'est la seule condition pertinente pour déterminer les variables d'intérêt. Ceci illustre le fait qu'appliquer l'opérateur ρ^{jst} vis à vis de toutes les variables d'un CN n'a pas d'intérêt : puisque l'on obtient l'ensemble des décisions prises (ici $w = 3$), l'état partiel ne pourra jamais être rencontré ou dominé plus tard sans redémarrage. L'application de $\rho^{jst \odot ent}$ est plus intéressante. Une fois que ρ^{ent} a été appliqué, on obtient l'état partiel Δ_1 dont les variables sont $\{x, y, z\}$. On peut appliquer ρ^{jst} pour déterminer quelles variables de Δ_1 ont un domaine qui peut être déduit des autres variables de Δ_1. La variable x est la seule variable pour laquelle toutes les valeurs supprimées ne peuvent pas être justifiées par les contraintes impliquant des variables *internes* à Δ_1 : $just(x \neq 3)$ implique une variable *externe*. Ceci est directement visible avec le graphe

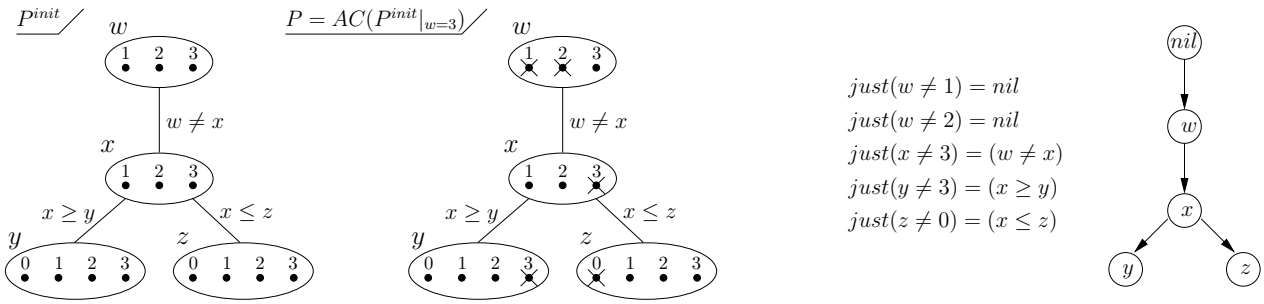


FIG. 7 – Gérer des justifications pendant la recherche.

$$P^{init} : \left\{ \begin{array}{l} w \in \{1,2,3\} \\ x \in \{1,2,3\} \\ y \in \{0,1,2,3\} \\ z \in \{0,1,2,3\} \end{array} \right\} \qquad P = AC(P^{init}|_{w=3}) : \left\{ \begin{array}{l} w \in \{3\} \\ x \in \{1,2\} \\ y \in \{0,1,2\} \\ z \in \{1,2,3\} \end{array} \right\}$$

$$\Delta_1 = \rho^{ent}(P) = \left\{ \begin{array}{l} x \in \{1,2\} \\ y \in \{0,1,2\} \\ z \in \{1,2,3\} \end{array} \right\}$$

$$\Delta_2 = \rho^{jst}_{vars(P)}(P) = \{\ w \in \{3\}\ \}$$

$$\Delta_3 = \rho^{jst \odot ent}(P) = \rho^{jst}_{vars(\Delta_1)}(P) = \{\ x \in \{1,2\}\ \}$$

FIG. 8 – États courants de P^{init} et P de la figure 7, et états partiels extraits en utilisant ρ^{ent}, ρ^{jst} et $\rho^{jst \odot ent}$.

de dépendance sur la figure 7. Nous obtenons alors l'état partiel $\Delta_3 = \{y \in \{1,2\}\}$.

La complexité spatiale pour l'enregistrement des justifications est $O(nd)$ alors que la complexité temporelle pour gérer cette structure est $O(1)$ même si une valeur est supprimée ou restaurée pendant la recherche.

8 Conclusion

Dans ce papier, nous avons brièvement montré les connexions existantes entre différentes approches d'apprentissage à l'aide d'un cadre général : celui des états partiels inconsistants. Deux opérateurs originaux permettant l'extraction d'IPSs ont également été décrits (et testés expérimentalement dans [12]). L'identification et l'exploitation d'IPSs sont deux activités prometteuses au regard des résultats déjà obtenus [10, 16, 13, 12].

Références

[1] R. Backofen and S. Will. Excluding symmetries in constraint based search. In *Proc. of CP'99*, pp. 73–87, 1999.

[2] R.R. Baker, F. Dikker, F.Tempelman, and P.M. Wognum. Diagnosing and solving over-determined constraint satisfaction problems. In *Proc. of IJCAI'93*, pp. 276–281, 1993.

[3] C. Bessiere. Arc-consistency in dynamic constraint satisfaction problems. In *Proc. of AAAI'91*, pp. 221–226, 1991.

[4] C. Bessiere. Constraint propagation. In *Handbook of Constraint Programming*, chapter 3. Elsevier, 2006.

[5] T. Fahle, S. Schamberger, and M. Sellman. Symmetry breaking. In *Proc. of CP'01*, pp. 93–107, 2001.

[6] F. Focacci and M. Milano. Global cut framework for removing symmetries. In *Proc. of CP'01*, pp. 77–92, 2001.

[7] E.C. Freuder and P.D. Hubbe. Using inferred disjunctive constraints to decompose CSPs. In *Proc. of IJCAI'93*, pp. 254–261, 1993.

[8] I. Gent and B. Smith. Symmetry breaking during search. In *Proc. of ECAI'00*, pp. 599–603, 2000.

[9] G. Katsirelos and F. Bacchus. Unrestricted nogood recording in CSP search. In *Proc. of CP'03*, pp. 873–877, 2003.

[10] G. Katsirelos and F. Bacchus. Generalized nogoods in CSPs. In *Proc. of AAAI'05*, pp. 390–396, 2005.

[11] C. Lecoutre and O. Roussel. Cohérences basées sur les valeurs en échec. In *Proc. of JFPC'09*, 2009.

[12] C. Lecoutre, L. Sais, S. Tabary, and V. Vidal. Exploiting past and future : Pruning by inconsistent partial state dominance. In *Proc. of CP'07*, pp. 453–467, 2007.

[13] C. Lecoutre, L. Sais, S. Tabary, and V. Vidal. Transposition Tables for CSPs. In *Proc. of AAAI'07*, pp 243–248, 2007.

[14] J.F. Puget. Symmetry breaking revisited. *Constraints*, 10(1) :23–46, 2005.

[15] I. Razgon and A. Meisels. Maintaining dominance consistency. In *Proc. of CP'03*, pp. 945–949, 2003.

[16] I. Razgon and A. Meisels. A CSP search algorithm with responsibility sets and kernels. *Constraints*, 12(2) :151–177, 2007.

[17] G. Richaud, H. Cambazard, B. O'Sullivan, and N. Jussien. Automata for nogood recording in CSPs. In *Proc. of SAT/CP workshop held with CP'06*, 2006.

[18] T. Schiex and G. Verfaillie. Nogood recording for static and dynamic CSPs. *IJAIT*, 3(2) :187–207, 1994.

[19] L. Zhang, C.F. Madigan, M.W. Moskewicz, and S. Malik. Efficient conflict driven learning in a Boolean satisfiability solver. In *Proc. of ICCAD'01*, pp. 279–285, 2001.

[20] L. Zhang and S. Malik. The quest for efficient Boolean satisfiability solvers. In *Proc. of CADE'02*, pp. 295–313, 2002.

Algorithme de décomposition de domaine pour la satisfaction et l'optimisation de contraintes

Wady Naanaa [1] **Maher Helaoui** [2] **Béchir Ayeb** [2]

[1] Faculté des sciences, Université de Monastir, Tunisie
[2] Pôle de Recherche en Informatique du Centre (PRINCE), Institut Supérieur d'Informatique et des Techniques de Communication, Route principale n°1 – 4011 Hammam Sousse
wnaanaa@planet.tn maher.helaoui@gmail.com ayeb_b@yahoo.com

Résumé

La notion de substituabilité directionnelle est une forme faible de la substituabilité de voisinage [6] qui a été proposée dans [17] pour améliorer la résolution de problèmes de satisfaction de contraintes (CSP) binaires. On part du fait que même si deux valeurs ne sont pas voisinage substituables, elles peuvent l'être si on restreint le voisinage en se référant à un ordre sur les variables. La substituabilité directionnelle permet de décomposer les domaines de valeurs des variables en de sous-ensembles de valeurs qui peuvent être essayées simultanément lors de la résolution du problème par un algorithme du type "*Backtracking*".

Dans le présent article, nous proposons deux extensions au travail présenté dans [17] :

- Tout d'abord, nous généralisons davantage la notion de substituabilité directionnelle en considérant, comme référence, une orientation du graphe d'inconsistance au lieu d'un ordre sur les variables.
- Ensuite, nous introduisons des conditions supplémentaires de substituabilité garantissant l'optimalité de la solution lors de la résolution de problèmes de satisfaction et d'optimisation de contraintes (CSOP).

Des résultats de simulation, sur plusieurs CSOPs modélisant des problèmes d'ordonnancement, montrent qu'une variante de l'algorithme du *Branch-and-Bound* qui exploite la substituabilité directionnelle est souvent plus efficace que l'algorithme original.

1 Introduction

Les problèmes de satisfaction de contraintes (CSPs) constituent un cadre simple et assez général pour modéliser divers problèmes combinatoires. Un CSP se définit par la donnée d'un ensemble de variables, chaque variable pouvant prendre une valeur parmi un ensemble de valeurs possibles appelé domaine. Puis, par un ensemble de contraintes exprimant des restrictions sur les combinaisons de valeurs permises. Résoudre un CSP revient à affecter aux variables, des valeurs de leurs domaines respectifs de telle sorte que toutes les contraintes soient satisfaites. Cette tâche est difficile du point de vue de la complexité puisqu'il s'agit de résoudre un problème NP-complet. En général, un CSP n'admet pas une solution unique. Pour les CSPs classiques, toutes les solutions sont supposées avoir la même "qualité". Ce qui fait que la résolution de tels problèmes se résume souvent à trouver la première solution. Cependant, dans plusieurs situations réelles telles que l'ordonnancement dans le secteur industriel, les solutions ne peuvent pas être considérées comme équivalentes dans le sens que certaines sont jugées meilleures que d'autres selon un critère bien déterminé. Il s'agit désormais de trouver la solution qui satisfait les contraintes et qui, en plus, optimise un critère mesurant la qualité des solutions. Les problèmes ainsi définis sont alors des problèmes de satisfaction de contraintes avec optimisation (CSOPs). Pour définir un CSOP et en plus des composantes définissant le CSP sous-jacent, on introduit une fonction coût qui permet d'exprimer des préférences entres les combinaisons de valeurs qui satisfont aux contraintes.

Dans cet article nous nous intéressons aux méthodes de résolution de CSOPs dites complètes : celles qui permettent de trouver une solution optimale si cette dernière existe. L'algorithme du *Branch-and-bound* [13] est sans doute l'algorithme complet le plus utilisé pour résoudre les CSOPs. C'est une procédure

d'exploration en profondeur d'abord munie d'une fonction heuristique servant à évaluer le coût de la solution complète que l'on peut espérer atteindre par l'extension de la solution partielle courante. Le rôle de la fonction heuristique est d'éviter l'exploration de branches de l'arbre de recherche qui ne mènent pas à des solutions meilleures que celles déjà trouvées.

Par ailleurs, la notion de substituabilité de voisinage [6] a été proposée et utilisée comme un moyen de filtrage des domaines de valeurs des variables des CSPs binaires. Plusieurs variantes de cette notion ont été proposées, parmi lesquelles la notion substituabilité directionnelle [17]. Cette dernière est une forme faible de la substituabilité de voisinage qui, au départ, visait à améliorer la résolution des CSPs binaires. On part de l'idée que même si deux valeurs ne sont pas voisinage substituables, elles peuvent l'être si on restreint le voisinage en se référant à un ordre sur les variables. La substituabilité directionnelle ne peut pas être utilisée comme un moyen de filtrage des domaines des variables comme c'est le cas pour la substituabilité de voisinage. Toutefois, on peut l'utiliser comme un moyen de décomposition des domaines des variables en de chaines de valeurs directionnellement substituables qui peuvent être considérées simultanément lors de la résolution des problèmes.
Dans le présent article, nous proposons deux extensions du travail présenté dans [17] :

- Nous généralisons davantage la notion de substituabilité directionnelle en considérant, comme référence, une orientation du graphe d'inconsistance au lieu d'un ordre sur les variables.
- Nous introduisons des conditions supplémentaires de substituabilité garantissant l'optimalité de la solution lors de la résolution de problème de satisfaction et d'optimisation de contraintes (CSOP).

Cet article est organisé comme suit : dans la section 2, nous donnons les définitions et les notations utilisées. Nous définissons par la suite (section 3), la notion de substituabilité directionnelle. On présente, dans la même section, une version de l'algorithme du Branch-and-Bound qui exploite la substituabilité directionnelle ainsi qu'un algorithme qui décompose les domaines de valeurs en de chaînes de valeurs directionnellement substituables. La section 4 présente un algorithme d'orientation du graphe d'inconsistance utilisée dans la détermination de la substituabilité directionnelle. Dans la section 5, nous reportons les résultats d'une expérimentation montrant l'apport de l'exploitation de la notion de substituabilité directionnelle pour la résolution de CSOPs binaires.

2 Définitions et notations

En plus des composantes qui définissent le problème de satisfaction de contrainte sous-jacent, un CSOP [23] fait intervenir une fonction coût qui permet de mesurer la qualité de chaque solution. Formellement, un CSOP peut être défini comme suit :

Définition 1 *Un problème de satisfaction et d'optimisation de contraintes (CSOP) est défini par un quadruplet* (X, D, C, Z) *tel que :*

1. $X = \{x_1, ..., x_n\}$ *est l'ensemble des variables.*
2. $D = \{D_1, ..., D_n\}$ *est l'ensemble des domaines de valeurs,* D_k *étant le domaine de* x_k*.*
3. $C = \{C_1, ..., C_m\}$ *est l'ensemble des contraintes. Chaque contrainte* C_k *implique un sous-ensemble de variables* $Var(C_k) = x_{k_1}, ..., x_{k_r}$ *appelé la portée de* C_k *et est définie par la relation, r-aire* $Rel(C_k) \subseteq D_{k_1} \times D_{k_2} ... \times D_{k_r}$ *contenant les r-uplets de valeurs respectant la contrainte* C_k*.*
4. $Z : \prod_{i=1}^{n} D_i \longrightarrow \mathbb{R}$ *est une fonction coût à minimiser.*

L'arité d'une contrainte est la taille de sa portée. L'arité d'un problème est l'arité maximale de ses contraintes. Dans cet article, nous nous limitons aux CSOPs binaires. Deux variables x_i et x_j reliées par une contrainte binaire, notée $C_{i,j}$, sont dites voisines. La valeur $a \in D_i$, dénotée aussi (x_i, a), est compatible avec $b \in D_j$ si $(a, b) \in Rel(C_{i,j})$. Dans ce cas, on dit que b est un support de a. Si chaque valeur du problème a, au moins, un support dans le domaine de chaque variable voisine alors le problème est dit arc-consistant. Le graphe d'inconsistance d'un CSOP binaire est un graphe simple dans lequel les sommets correspondent aux valeurs des variables et les arêtes relient les couples de sommets qui représentent des valeurs incompatibles. L'ensemble des valeurs incompatibles avec une valeur a d'une variable x_i, est défini par

$$N(x_i, a) = \{(x_j, b) \mid C_{i,j} \in C \text{ et } (a, b) \notin Rel(C_{i,j})\}$$

En se référant à [6], une valeur a d'une variable x_i est voisinage substituable (VS) à une valeur b de x_i si et seulement si $N(x_i, a) \subseteq N(x_i, b)$. Cette définition peut être étendue pour tenir compte de la fonction coût. Toutefois, cette dernière doit vérifier certaines propriétés pour que ceci soit possible. En effet, la fonction coût est une fonction globale (n-aire) puisqu'elle s'applique à toutes les variables du problème (voir définition 1) alors que la notion de substituabilité de voisinage est une notion locale. On propose donc que Z soit une fonction telle qu'il est possible de trouver une fonction coût unaire $z : \bigcup_{i=1}^{n} D_i \longrightarrow \mathbb{R}$ qui vérifie

$$\forall\ T \in D_1 \times \ldots \times D_n, \quad Z(T) = \oplus_{i=1}^{n} z(T_i) \qquad (1)$$

où $\oplus$ désigne un opérateur monotone sur $\mathbb{R}$ et T_i la valeur affectée à la variable x_i dans T. La substituabilité de voisinage pour les CSOPs binaires peut alors être définie de la manière suivante :

$$a \preceq b \quad \Leftrightarrow \quad \left\{ \begin{array}{l} N(x_i, a) \subseteq N(x_i, b) \quad \text{et} \\ z(x_i, a) \leq z(x_i, b) \end{array} \right.$$

Il en découle qu'à partir de toute solution dans laquelle x_i a pour valeur b, on peut déduire une solution de qualité, au moins, égale si l'on substitue a à b. Par conséquent, b peut être supprimée du domaine de x_i sans que l'on perde toutes les solutions optimales du problème.

3 Substituabilité directionnelle (DS)

3.1 Définitions et propriétés

La substituabilité directionnelle [17] est une forme faible de substituabilité de voisinage. Au départ, cette notion a été définie en se référant à un ordre total sur les variables du problème. Dans le présent article, on généralise la notion de substituabilité directionnelle en utilisant, comme référence, une orientation du graphe d'inconsistance du CSOP. Formellement, une orientation Λ du graphe d'inconsistance d'un CSOP est une affectation d'une direction à chaque arête $\{a, b\}$ du graphe donnant lieu, ou bien à l'arc (a, b) ou bien à l'arc (b, a). Etant donnée une orientation Λ du graphe d'inconsistance d'un CSOP, on définit l'ensemble des conflits directionnels d'une valeur a d'une variable x_i par rapport à Λ de la manière suivante :

$$\begin{array}{rcl} \vec{N}(x_i, a) & = & \{(x_j, b) \mid\ C_{i,j} \in C,\ (a, b) \notin Rel(C_{i,j}) \text{ et} \\ & & (a, b) \in\ \Lambda\} \end{array} \qquad (2)$$

La substituabilité directionnelle se définit alors comme suit :

Définition 2 *Soit $\mathcal{P} = (X, D, C, Z)$ un CSOP binaire et a et b deux valeurs d'une variable x_i de $\mathcal{P}$. a est dite directionnellement substituable à b par rapport à une orientation Λ du graphe d'inconsistance de $\mathcal{P}$, (notation : $a \preceq_\Lambda^{\mathcal{P}} b$) si et seulement si*

$$\vec{N}(x_i, a) \subseteq \vec{N}(x_i, b) \quad et \quad z(x_i, a) \leq z(x_i, b) \qquad (3)$$

Dans ce qui suit, on utilisera la notation $\preceq_\Lambda$ au lieu de $\preceq_\Lambda^{\mathcal{P}}$ si ceci n'entraine pas d'ambiguïté. La relation $\preceq_\Lambda$ définit un préordre sur le domaine de chaque variable. Ainsi, chaque domaine D_i peut être subdivisé en de sous-ensembles $D_i = D_{i,1} \cup \ldots \cup D_{i,s}$ tel que les éléments de chaque $D_{i,k},\ k = 1, ..., s$ sont tous deux à deux comparables. C'est-à-dire, que pour tout $a, b \in D_{i,k}$, on a $a \preceq_\Lambda b$ ou $b \preceq_\Lambda a$. Chaque $D_{i,k}$ est donc une chaine de valeurs totalement ordonnée par $\preceq_\Lambda$. Dans chaque chaine $D_{i,k}$, on peut distinguer le sous-ensemble des éléments minimaux.

$$\min(D_{i,k}) = \{a \in D_{i,k} \mid \forall\ b \in D_{i,k},\ a \preceq_\Lambda b\} \qquad (4)$$

La proposition suivante identifie une classe polynomiale de CSOPs.

Proposition 1 *Soit $\mathcal{P} = (X, D, C, Z)$ un CSOP binaire arc-consistant et soit Λ est une orientation du graphe d'inconsistance de $\mathcal{P}$. Si chacun des domaines de valeurs de $\mathcal{P}$ est une chaine non vide de valeurs directionnellement substituables par rapport à Λ alors une solution optimale de $\mathcal{P}$ peut être trouvée en un temps polynomial.*

Preuve : On montre qu'en sélectionnant un élément minimum de chaque domaine de valeurs, on obtient une solution optimale. Ce qui veut dire que tout n-uplet $T \in \min(D_1) \times \ldots \times \min(D_n)$ est une solution optimale. En se référant à (3) et au fait que la fonction z est calculable en temps polynomial, cette sélection peut être effectuée en temps polynomial.

Supposons que $T \in \min(D_1) \times \ldots \times \min(D_n)$ n'est pas une solution optimale. Ceci implique que T est inconsistant ou que $Z(T)$ n'est pas minimum.

Commençons par supposer que T est inconsistant. Donc T doit contenir, au moins, une paire de valeurs incompatibles. Soit $a \in \min(D_i)$ et $b \in \min(D_j)$ une telle paire. Puisque a et b sont incompatibles, on doit avoir $(a, b) \in \Lambda$ ou $(b, a) \in \Lambda$. Supposons, sans perte de généralité, que $(a, b) \in \Lambda$, (sinon on pourra raisonner sur b au lieu de a et obtenir le même résultat). Il s'ensuit que $b \in \vec{N}(x_i, a)$, et puisque $a \in \min(D_i)$ alors pour tout $a' \in D_i$, on doit avoir $b \in \vec{N}(x_i, a')$. Ce qui veut dire que b n'a pas de support dans D_i et donc que $\mathcal{P}$ n'est pas arc-consistant, d'où une contradiction.

Supposons, à présent que $Z(T)$ n'est pas minimum, donc, qu'il existe $T' \in D_1 \times \ldots \times D_n$ tel que $Z(T') < Z(T)$. Puisque les valeurs de la fonction Z sont obtenues à partir de celles de z en utilisant un opérateur monotone (voir équation (1)), il doit exister $x_i \in X$ tel que $T_i = (x_i, a), T_i' = (x_i, a')$ et $z(x_i, a') < z(x_i, a)$. Il en résulte que $a \notin \min(D_i)$, d'où une contradiction.

Nous nous intéressons aux méthodes de résolution de CSOPs dites complètes : celles qui permettent de trouver une solution optimale si cette dernière existe. L'algorithme du Branch-and-bound est l'algorithme complet le plus communément utilisé pour résoudre les CSOPs. Cet algorithme explore l'espace de recherche du problème à résoudre en effectuant une recherche arborescente en profondeur d'abord. Les problèmes considérés tout lelong d'un chemin de l'arbre de recherche sont des réductions du problème initial que l'on peut définir de la manière suivante :

Définition 3 *Un problème $\mathcal{P} = (X, D, C, Z)$ est une réduction d'un problème $\mathcal{P}' = (X', D', C', Z')$ (notation $\mathcal{P} \sqsubseteq \mathcal{P}'$) si et seulement si*
- $X = X'$,
- $D_i \subseteq D'_i$, pour tout $x_i \in X$,
- $C = \{C_{i,j} \mid C'_{i,j} \in C'$ et $Rel(C_{i,j}) = Rel(C'_{i,j}) \cap D_i \times D_j\}$,
- Z est la restriction de Z' à $D_1 \times \ldots \times D_n$.

Une propriété essentielle de la substituabilité directionnelle est qu'elle soit préservée quand on passe d'un problème à l'une de ses réductions. En effet, on a

Proposition 2 *Soient $\mathcal{P} = (X, D, C, Z)$ et $\mathcal{P}' = (X, D', C', Z)$ deux CSOPs binaires tels que $\mathcal{P} \sqsubseteq \mathcal{P}'$ et Λ, (resp. Λ') une orientation du graphe d'inconsistance de $\mathcal{P}$ (resp. de $\mathcal{P}'$) telle que $\Lambda \subseteq \Lambda'$. Alors, on a*

$$\forall\ x_i \in X,\ \forall\ a, b \in D_i \cap D'_i, \quad a \preceq^{\mathcal{P}'}_{\Lambda'} b \Rightarrow a \preceq^{\mathcal{P}}_{\Lambda} b$$

Preuve : Désignons par $\vec{N}^{\mathcal{P}}(x_i, a)$ l'ensemble des conflits directionnels de (x_i, a) dans $\mathcal{P}$ et par ΔD l'ensemble des valeurs de $\mathcal{P}'$ qui ne figurent pas dans $\mathcal{P}$. On a donc $\Delta D = \bigcup_{x_i \in X} D'_i - D_i$. Soient (x_i, a) et (x_i, b) deux valeurs quelconques disponibles dans $\mathcal{P}$ et $\mathcal{P}'$. Puisque $\mathcal{P} \sqsubseteq \mathcal{P}'$, on a

$$\vec{N}^{\mathcal{P}}(x_i, a) = \vec{N}^{\mathcal{P}'}(x_i, a) \ - \Delta D \tag{5}$$

De même

$$\vec{N}^{\mathcal{P}}(x_i, b) = \vec{N}^{\mathcal{P}'}(x_i, b) \ - \Delta D \tag{6}$$

D'autre part, en partant de $a \preceq^{\mathcal{P}'}_{\Lambda} b$, on déduit que $\vec{N}^{\mathcal{P}'}(x_i, a) - \Delta D \subseteq \vec{N}^{\mathcal{P}'}(x_i, b) - \Delta D$ et que $z(x_i, a) \leq z(x_i, b)$. D'après (5) et (6), on obtient $\vec{N}^{\mathcal{P}}(x_i, a) \subseteq \vec{N}^{\mathcal{P}}(x_i, b)$. D'où, $a \preceq^{\mathcal{P}}_{\Lambda} b$.

Une conséquence directe de la proposition 2 est que pour toute paire de CSOPs $\mathcal{P}$ et $\mathcal{P}'$ telles que $\mathcal{P} \sqsubseteq \mathcal{P}'$, si D'_i est une chaine de valeurs DS dans $\mathcal{P}'$ alors D_i est une chaine de valeurs DS dans $\mathcal{P}$.

3.2 Exemple

Considérons le problème Job-Shop classique décrit dans la figure 1. Il s'agit de planifier l'exécution de deux jobs J_1 et J_2 sur trois machines M_1, M_2 et M_3. Chaque job J_i se compose de trois tâches $T_{i,1}, T_{i,2}$ et $T_{i,3}$. Une tâche $T_{i,j}$ est dédiée à être exécutée sur une machine unique. Pour cet exemple, on suppose que la durée de toutes les tâches est fixée à une unité de temps, sauf $T_{2,2}$ qui consomme deux unités. Les job-shops font intervenir deux types de contrainte : des contraintes de précédence qui imposent que les tâches d'un même job soient exécutées les une après les autres ainsi que des contraintes de ressource qui empêchent qu'une machine exécute plus d'une tâche à la fois. Résoudre un tel problème revient à affecter un temps de début d'exécution à chacune des tâches de façon à satisfaire les contraintes de précédence et de ressource toute en minimisant le temps total d'exécution de toutes les tâches (makespan).

Une modélisation possible des problèmes job-shops en termes de CSOP binaire consiste à associer une variable du CSOP à chacune des tâches. Ainsi, pour notre exemple, on obtient un problème impliquant six variables $x_1, \ldots, x_6$. Les domaines de valeurs des variables représentent une discrétisation du temps maximum impartie à l'exécution de toutes les tâches. Pour le présent exemple, on suppose que les différentes tâches peuvent commencer à des instants représentés par les entiers de 0 à 4. Les contraintes impliquées sont des contraintes binaires de précédence et de ressource. On en trouve en tout sept contraintes (voir figure 1). La fonction coût à minimiser est définie par

$$Z(T) = \max_{1 \leq i \leq 6} \{a + durée(x_i) \mid T_i = (x_i, a)\}$$

où T désigne une instanciation de toutes les variables du CSOP. En tenant compte des contraintes de précédence, la fonction Z peut être calculée en prenant le maximum uniquement sur la paire $\{x_3, x_6\}$ au lieu de X. On en déduit une définition possible de la fonction z

$$z(x_i, a) = \begin{cases} a + durée(x_i) & \text{si } i \in \{3, 6\} \\ 0 & \text{sinon} \end{cases} \tag{7}$$

On obtient donc $Z(T) = \max_{i=1}^{6}(z(T_i))$, où max est bien un opérateur monotone. Après application d'un algorithme d'arc-consistance, on obtient le problème dont le graphe d'inconsistance est représenté dans la figure 2. Dans cette même figure on peut également voir une orientation du graphe d'inconsistance. En se référant à cette orientation, on obtient les ensembles des conflits directionnels donnés dans le tableau 1. En examinant ce tableau, on constate que

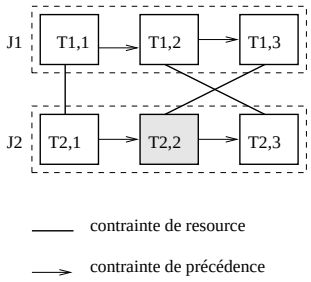


FIGURE 1 – Un problème du type job-shop comportant deux jobs et six tâches. Toutes les tâches sont supposées avoir une durée d'une unité de temps, sauf $T_{2,2}$ qui en consomme deux.

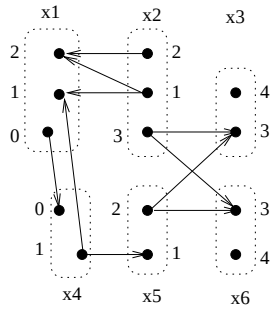


FIGURE 2 – Graphe d'inconsistance du job-shop décrit dans la figure 1 après application d'un algorithme d'arc-consistance. On peut également voir une orientation du graphe d'inconsistance.

tous les domaines de valeurs sont des chaines de valeurs DS sauf D_2 qui est composé de deux chaines : $\{(x_2,1),(x_2,2)\}$ et $\{(x_2,3)\}$. En réduisant le domaine de x_2 à $\{(x_2,1),(x_2,2)\}$ puis à $\{(x_2,3)\}$, on obtient deux sous-problèmes qui, d'après la proposition 2, ne contiennent que des chaines de valeurs DS comme domaine. D'après la proposition 1, ces deux sous-problèmes peuvent être résolus en un temps polynomial.

3.3 Exploitation de la Substituabilité Directionnelle

La substituabilité directionnelle peut être utilisée pour améliorer la résolution de CSOPs binaires et ceci en l'intégrant à l'algorithme du Branch-and-Bound pour obtenir l'algorithme BAB–DS (voir fonction 1). Pour avoir une méthode de résolution plus efficace, BAB–DS intègre également une procédure qui permet de maintenir l'arc-consistance durant la recherche comme dans l'algorithme de résolution de CSP MAC [19].

BAB–DS prend comme paramètres le CSOP à

	0	1	2	3	4
x_1	$(x_4,0)$	$\emptyset$	$\emptyset$		
x_2		$(x_1,1)$ $(x_1,2)$	$(x_1,2)$	$(x_3,3)$ $(x_6,3)$	
x_3				$\emptyset$	$\emptyset$
x_4	$\emptyset$	$(x_1,1)$ $(x_5,1)$			
x_5		$\emptyset$	$(x_3,3)$ $(x_6,3)$		
x_6				$\emptyset$	$\emptyset$

TABLE 1 – Conflits directionnels des valeurs arc-consistantes du CSOP décrit dans la figure 2.

résoudre (X,D,C,Z), (qui est supposé être arc-consistant), une orientation du graphe d'inconsistance du problème Λ, l'ensemble des variables non encore instanciées Y et la meilleure solution courante I^* et procède comme suit : Il sélectionne une variable non encore instanciée, calcule la partition de son domaine de valeurs en des chaines de valeurs DS et décompose le problème courant en deux sous-problèmes. Le premier sous-problème est obtenu en réduisant le domaine de valeurs de la variable courante à une chaine de valeurs DS (notée $D_{i,k}$ dans le pseudo-code). L'arc-consistance du problème résultant est alors restauré en utilisant un algorithme d'arc-consistance. Puis, un appel récursif est effectué pour considérer les variables restantes. Cet appel permet d'obtenir la meilleure solution du sous-problème qui limite les valeurs possibles de la variable courante aux éléments de $D_{i,k}$. Ensuite, un processus de restauration des domaines de valeurs est effectué et $D_{i,k}$ est éliminée du domaine de la variable courante. Après restauration de l'arc-consistance du problème résultant, l'algorithme effectue un deuxième appel récursif pour considérer les autres chaines de valeurs DS. Les deux appels récursifs ne sont effectués que si une fonction heuristique (h) indique qu'il est possible d'obtenir des solutions de qualité meilleures que celles déjà trouvées. Si l'algorithme réussit à instancier toutes les variables alors il exécute un processus polynomial (ligne 4) pour extraire la meilleure solution à partir des chaines sélectionnées.

Fonction 1 BAB–DS$((X,D,C,Z),\Lambda,Y,I^*)$

1 : **si** $Y=\emptyset$ **alors**
2 : retourner(min(D))
3 : **sinon**
4 : $x_i \leftarrow$ Sélect(Y)
5 : $D_i \leftarrow$ DS-Partition$(D_i,\Lambda,(X,D,C,Z))$
6 : $D_{i,k} \leftarrow$ Select (D_i)
7 : $D_i \leftarrow D_{i,k}$
8 : AC(X,D,C)
9 : **si** $\emptyset \notin D$ **et** $h(D) < Z(I)$ **alors**

10 : $I \leftarrow$BAB–DS$(Y - \{x_i\}, (X, D, C, Z), \Lambda, I^*)$
11 : **si** $Z(I) < Z(I^*)$ **alors** $I^* \leftarrow I$
12 : Restaurer (D)
13 : $D_i \leftarrow D_i$ - $D_{i,k}$
14 : AC(X, D, C)
15 : **si** $\emptyset \notin D$ **et** $h(D) < Z(I^*)$ **alors**
16 : $I \leftarrow$BAB–DS$(Y - \{x_i\}, (X, D, C, Z), \Lambda, I^*)$
17 : **si** $Z(I) < Z(I^*)$ **alors** $I^* \leftarrow I$
18 : Restaurer(D)
19 : retourner(I^*)

3.4 Algorithme de DS partition

La relation $\preceq_\Lambda$ définit un préordre sur le domaine de chaque variable du problème. Ce préordre peut être utilisé pour partitionner les domaines de valeurs des variables en de chaines de valeurs DS. En effet, à partir de $\preceq_\Lambda$, on définit, tout d'abord, la relation $\sim_\Lambda$ telle que $a \sim_\Lambda b$ si et seulement si $a \preceq_\Lambda b$ et $b \preceq_\Lambda a$. On peut facilement vérifier que $\sim_\Lambda$ est une relation d'équivalence. $\preceq_\Lambda$ induit un ordre partiel $\leq_\Lambda$ sur l'ensemble $D_i \backslash \sim_\Lambda$ des classes d'équivalence de $\sim_\Lambda$ tel que $[a] \leq_\Lambda [b]$ si et seulement si $a \preceq_\Lambda b$, où $[a]$ désigne la classe d'équivalence de la valeur a.

En général, étant donné un ensemble partiellement ordonné E, on peut avoir plusieurs partitions de E en chaines d'éléments totalement ordonnés. En théorie des ensembles, la partition optimale en chaines d'un ensemble partiellement ordonné est connue sous le nom de partition en chaines de Dilworth (DCP) [5]. C'est une partition qui contient le nombre minimum de chaines parmi toutes les partitions possibles. La taille d'une telle partition (i.e., le nombre de chaines de la partition), définit la largeur de l'ordre partiel. Le problème qui consiste à trouver la DCP d'un ensemble partiellement ordonné peut être résolu en temps polynomial en l'exprimant comme un problème de recherche d'un couplage maximum dans un graphe bipartite [9].

Motivé par le fait qu'à chaque noeud de l'arbre de recherche, l'algorithme de résolution aura autant de choix que de chaines dans une DCP du domaine de la variable courante, nous proposons de calculer une DCP à chaque noeud de l'arbre de recherche. Par cette stratégie, on cherche à minimiser la taille de l'arbre de recherche que l'algorithme aura à explorer.

La première étape du calcul de la DCP (voir la fonction 2) consiste à déterminer la relation $\sim_\Lambda$. Ceci peut être accompli en $O(nd^2)$ étapes en utilisant l'algorithme décrit dans [6] ou celui proposé dans [17], n et d étant respectivement, le nombre de variables du problème et la taille maximale des domaines de valeurs. On en déduit les classes d'équivalences $D_i/\sim_\Lambda$ en $O(d)$ étapes. L'étape la plus couteuse est celle du calcul de la relation $\leq_\Lambda$. Au pire des cas, $d(d-1)/2$ tests d'inclusion entre des paires d'ensembles de conflits directionnels sont nécessaires. Chaque test d'inclusion peut être accompli en $O(nd)$ puisque chaque ensemble de conflit directionnel contient au maximum $d(n-1)$ éléments. D'où une complexité de $O(nd^3)$ pour cette étape. Ensuite, l'algorithme construit un graphe bipartite $G = (V, U, E)$ tel que $V = U = D_i/\sim_\Lambda$ et l'ensemble des arêtes E contient une arête $([a], [b])$ si et seulement si $[a] \leq_\Lambda [b]$. La construction de G nécessite $O(d^2)$ étapes. Un algorithme de couplage maximum est alors appliqué à G. On utilise l'algorithme décrit dans [9] qui s'exécute en $O(d^{2.5})$ étapes. Les chaines de la DCP sont extraites du couplage maximum en incluant les éléments de $[a]$ et ceux de $[b]$ dans une même chaine chaque fois que l'arête $([a], [b])$ fait partie du couplage maximum. Ceci demande $O(d^2)$ étapes. D'où une complexité totale de $O(nd^3)$.

Fonction 2 DS–Partition$(D_i, \Lambda, (X, D, C, Z))$

1 : $\sim_\Lambda \longleftarrow$ DirInterchangeable$(D_i, \Lambda, (X, D, C, Z))$
2 : $D_i/\sim_\Lambda \longleftarrow$ ExtractEqClass$(\sim_\Lambda, D_i, (X, D, C, Z))$
3 : $\leq_\Lambda \longleftarrow$ DirSubstituable$(D_i/\sim_\Lambda, (X, D, C, Z))$
4 : $G \longleftarrow$ BipartiteGraph$(D_i/\sim_\Lambda, \leq_\Lambda)$
5 : $M \longleftarrow$ CouplageMaximum(G)
6 : $\mathcal{D}_i \longleftarrow$ ExtractChaines$(D_i, \sim_\Lambda, M)$
7 : return$(\mathcal{D}_i)$

3.5 Exemple (suite)

Reprenons l'exemple du paragraphe 3.2. Comme on l'a déjà constaté, après application d'un algorithme d'arc-consistance, tous les domaines de valeurs se trouvent réduits à des chaines de valeurs DS sauf D_2 qui est composé de deux chaines : $\{(x_2, 1), (x_2, 2)\}$ et $\{(x_2, 3)\}$. L'application de l'algorithme BAB–DS à cet exemple se résume à instancier les variables x_1, x_3, x_4, x_5 et x_6 par les seules chaines qui constituent leurs domaines respectifs. Pour x_2, il y a deux instanciations possibles qui correspondent aux deux chaines données ci-dessus. Commençons par réduire D_2 à la chaine $\{(x_2, 1), (x_2, 2)\}$. A ce stade, tous les domaines sont des chaines de valeurs DS. La restauration de l'arc-consistance (ligne 8) ne change rien à ce fait d'après la proposition 2. Le graphe d'inconsistance du problème obtenu est illustré dans la figure 3-(a). Par la suite, l'algorithme choisit un élément minimum de chacun des domaines (ligne 2). D'après le tableau 1 et la figure 3-(a), le choix est unique pour chacune des variables et l'on obtient l'instanciation complète $(1, 2, 3, 0, 1, 3)$ qui est bien une solution qui satisfait toutes les contraintes et qui a un coût égal à 4 unités de temps. Enfin, BAB–DS réduit D_2 à $\{(x_2, 3)\}$.

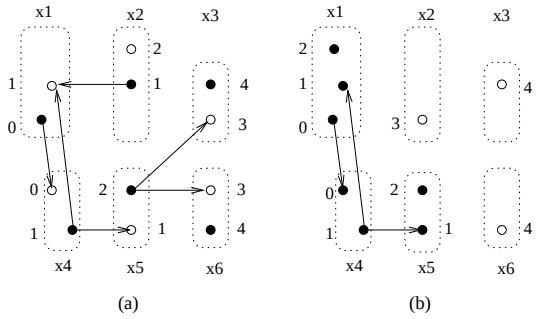


FIGURE 3 – (a) Graphe d'inconsistance du CSOP associé au job-shop décrit dans la figure 1 après réduction de D_2 à la chaine $\{(x_2, 1), (x_2, 2)\}$ et application d'un algorithme d'arc-consistance. (b) Graphe d'inconsistance du même CSOP après réduction de D_2 à la chaine $\{(x_2, 3)\}$ et application d'un algorithme d'arc-consistance.

En appliquant l'algorithme d'arc-consistance, puis la fonction heuristique h sur les domaines ainsi réduits, on déduit qu'il n'est pas possible de trouver une solution meilleure que celle déjà trouvée. En effet, les domaines de x_3 et de x_6 sont tous les deux réduits à la seule valeur 4 (voir figure 3-(b)). Ce qui veut dire que la meilleure solution qu'on peut espérer trouver a un coût de 5 unités de temps.

4 Orientation du graphe d'inconsistance

L'orientation du graphe d'inconsistance utilisée comme référence (voir équation 2) pour déterminer les valeurs DS a un grand impact sur l'occurrence des valeurs DS. Les meilleures résultats ont été obtenus lorsqu'on s'est référé à une orientation qui favorise l'occurrence des valeurs DS à des niveaux proches de la racine de l'arbre de recherche. Une telle orientation est obtenue dynamiquement en utilisant un algorithme glouton (voir fonction 3). Au départ du processus de recherche, l'orientation Λ est vide. A chaque étape, l'algorithme d'orientation ne considère que les arêtes qui relient les valeurs de la variable courante à ceux des variables déjà instanciées. Soit a une valeur quelconque de la variable courante qui est incompatible avec une valeur b d'une variable déjà instanciée. L'arête $\{a, b\}$ donnera lieu à l'arc (a, b), qui sera ajouté à Λ.

Cette stratégie favorise l'occurrence des valeurs DS à des niveaux proches de la racine. En effet, rappelons que seulement les arcs sortants d'une valeur sont pris en considération lors du calcul des conflits directionnels de cette valeur. En ne considérant que les arêtes qui vont vers les valeurs des variables instanciées, on augmente les chances des valeurs des variables traitées à des niveaux proches de la racine à être DS les une aux autres. La raison de ceci est que, proche de racine, il y a peu de variables instanciées. Le fait de considérer de larges chaines de valeurs DS à des niveaux peu profonds de l'arbre de recherche est une stratégie souvent avantageuse car, à ce stade, l'heuristique d'ordre des valeurs ne dispose pas d'assez d'information pour faire des choix de valeurs corrects. En prenant des décisions portant sur plusieurs valeurs, on peut éviter des erreurs qui, plus elles sont commises tôt, plus elles sont couteuses en temps d'exploration.

Fonction 3 Orientation$(x_i, \Lambda, Y, (X, D, C, Z))$

1 : **pour tout** $x_j \in X - Y \mid C_{i,j} \in C$ **faire**
2 : **pour tout** $a \in D_i$ **faire**
3 : **pour tout** $b \in D_j \mid (a, b) \notin Rel(C_{i,j})$ **faire**
4 : $\Lambda \longleftarrow \Lambda \cup \{(a, b)\}$
5 : Retourer(Λ)

5 Résultats expérimentaux

L'expérimentation porte sur la comparaison des performances de l'algorithme du Branch-and-Bound classique et de BAB–DS sur des problèmes du type job-shop [1] qui ont été modélisés comme indiqué dans le paragraphe 3.2. L'algorithme d'arc-consistance employé est AC-3 [16]. L'heuristique d'ordre de variables est *min-domain/wdeg* [2], Pour l'ordre des valeurs, nous avons testé deux heuristiques : *min. conflit* qui privilégie les valeurs qui ont le moins de conflits [7] et *min. valeur* qui tient compte de la fonction coût en favorisant les plus petites valeurs. La fonction z, qui est indispensable pour définir la substituabilité directionnelle, est déduite de Z de manière analogue à celle spécifiée par l'équation (7). Les critères d'évaluation des performances sont le temps de calcul et la qualité de la meilleure solution trouvée. Les deux algorithmes ont été implémentés en C++ et exécutés sur un PC à 2 GHZ de fréquence et une mémoire vive de 4GB. Les problèmes test utilisés sont ceux référencés par JS-TAILLARD-15 disponibles sur [14]. Ce sont dix instances de job-shops comportant 15 jobs et 15 machines, (donc 225 tâches), générés selon le modèle décrit dans [22]. Pour ces instances de job-shop, la discrétisation du temps maximum impartie pour l'exécution de toutes les tâches donne plus de mille valeurs. Pour obtenir des solutions en des temps d'exécution raisonnables pour des CSOPs impliquant de si large domaines de valeurs, on a élargi davantage les domaines de valeurs pour obtenir des instances plus fa-

ciles. Ainsi, on a multiplié les tailles des domaines originaux par un facteur de 105%, 104% puis 103% pour obtenir des instances qui vont des plus faciles aux plus difficiles. De plus, on a fait recours à une recherche par divergence limitée [8], pour les deux algorithmes.

Les résultats obtenus (voir figure 4) montrons clairement que BAB–DS a trouvé des solutions pour plus d'instances que BAB. Pour les instances auxquelles les deux algorithmes ont réussit à trouver des solutions, on constate que dans la plupart des cas, ou bien que la solution trouvée par BAB–DS a un coût meilleur que celle trouvée par BAB, ou bien que BAB–DS trouve une solution ayant le même coût mais en moins de temps. Avec l'heuristique min. conflit, sur les trente instances considérées, il y a uniquement trois instances (tai15-104-4, tai15-104-7 et tai15-103-7) sur lesquelles, BAB a été meilleur que BAB–DS. Avec l'heuristique min. valeur, BAB–DS a donné des résultats meilleures sur toutes les instances sauf l'instance tai15-104-9. Enfin, la supériorité de BAB–DS ne semble dépendre ni de la difficulté des instances ni de l'heuristique d'ordre des valeurs utilisée puisque BAB–DS est globalement meilleur sur les trois niveaux de difficulté.

6 Travaux Connexes

Plusieurs travaux ont porté sur l'exploitation dynamique de l'interchangeabilité de voisinage qui est un cas particulier de la substituabilité de voisinage [10, 15, 21]. Les algorithmes proposés détectent des sous-ensembles de valeurs voisinage interchangeables, dont le produit cartésien fournit une représentation compacte de l'ensemble de solutions du CSP. Cette représentation des solutions par produit cartésien est très utilisée pour le calcul de toutes les solutions. Dans [4], les auteurs montrent que les algorithmes s'appuyant sur la représentation par produit cartésien sont aussi efficaces pour la recherche de la première solution d'instances difficiles de CSP.

Toutefois, l'efficacité de ces méthodes reste tributaire de l'occurrence de l'interchangeabilité de voisinage. Cette dernière est plutôt rare quand il s'agit de situations réelles. Pour cette raison, plusieurs variantes de l'interchangeabilité de voisinage ont été proposées. Dans [24], les auteurs proposent la notion d'interchangeabilité conditionnelle, qui vise à détecter plus de valeurs voisinage interchangeables. Une condition est une restriction du domaine des variables voisines qui permet de capturer l'interchangeabilité dans des situations où l'interchangeabilité de voisinage ne s'applique pas. Bowen et Likitvivatanavong ont introduit le concept de transmutation de domaine [3]. Leur approche consiste à regrouper plusieurs valeurs afin d'exploiter plus intensivement l'interchangeabilité. Les auteurs ont reporté que leur approche est particulièrement avantageuse dans la recherche de toutes les solutions d'un CSP. Le filtrage par interchangeabilité de voisinage a été aussi appliqué aux CSPs non binaires [12]. Dans [18], les auteurs ont généralisé la définition d'interchangeabilité en vue de l'appliquer aux soft CSPs. Ils ont introduit deux relaxations en s'appuyant sur les notions de dégradation et de seuil. Ces deux formes d'interchangeabilités sont respectivement notées ${}^{\delta}$interchangeabilité de voisinage (${}^{\delta}$NI) et ${}_{\alpha}$interchangeabilité voisinage (${}_{\alpha}$NI). Il a été expérimentalement montré que les valeurs ${}^{\delta}$NI et ${}_{\alpha}$NI sont fréquemment rencontrées lors de la résolution de CSPs flous et de CSPs pondérés. Enfin, la notion de substituabilité a été également définie pour les contraintes où des tuples redondants sont détectés et supprimés des relations définissant les contraintes [11], .

Toutes les approches citées ci-dessus utilisent diverses formes d'interchangeabilité ou de substituabilié pour déterminer les valeurs ou les tuples qu'on peut supprimer sans perdre toutes les solutions (optimales) du problème. Ces méthodes peuvent donc être classées comme des méthodes de filtrage. Dans notre cas, la substituabilité directionnelle ne permet pas d'éliminer les valeurs ou les tuples redondants, mais permet, à un algorithme de recherche arborescente, d'effectuer des branchements sur plusieurs valeurs à la fois. La substituabilité directionnelle est donc une méthode de décomposition de domaine.

7 Conclusion

Dans cet article, nous avons proposé deux extensions à la notion de substituabilité directionnelle présentée par l'un des auteurs dans [17]. Tout d'abord, nous avons généralisé davantage la substituabilité directionnelle en considérant, comme référence, une orientation du graphe d'inconsistance du problème au lieu d'un ordre sur les variables du problème. Puis, nous avons introduit des conditions supplémentaires de substituabilité garantissant l'optimalité de la solution lors de la résolution de problème de satisfaction et d'optimisation de contraintes (CSOP).
Les résultats de simulation sur plusieurs CSOPs qui codent des problèmes d'ordonnancement du type jobshop ont montré qu'une variante de l'algorithme du Branch-and-Bound exploitant la substituabilité directionnelle est souvent plus efficace que l'algorithme original.

Une extension possible de ce travail consiste à proposer une formulation encore plus générale de la substituabilité directionnelle afin de pouvoir l'appliquer aux Soft CSPs [20] qui est un formalisme offrant une plus grande capacité à exprimer des problèmes réels.

Références

[1] Applegate, D., Cook, W. A computational study of the job-shop scheduling problem, *ORSA Journal on Computing* 3 : 149-156. 1991

[2] Boussemart, F., Hemery, F., Lecoutre, C., Sais, L. Boosting systematic search by weighting constraints. In *Proceedings of ECAI'04*, pages 146–150, 2004

[3] Bowen, J., Likitvivatanavong, C. Domain transmutation in constraint satisfaction problems. In *Proceedings of AAAI-04*, 2004

[4] Choueiry, B. Y., Davis, A. M. Dynamic Bundling : less effort for more solutions. In *Proceedings of the* 5^{th} *International Symposium on Abstraction, Reformulation and Approximation, Lecture Notes in Computer Science*, pages 64–82, volume 2371, Springer-Verlag, 2002

[5] Dilworth, R. P. A decomposition theorem for partially ordered sets. *Annals of Mathematics*, 51 : 161-166. 1950

[6] Freuder, E. C. Eliminating interchangeable values in constraint satisfaction problems. In *Proceedings of AAAI-91*, pages 227–233, 1991

[7] Frost, D., Dechter, R. Look-ahead value ordering for constraint satisfaction problems. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 572–578, 1995

[8] Harvey, W., Ginsberg, M. Limited discrepancy search. In *Proceedings of the 14th International Joint Conference On Artificial Intelligence*, pages 607–613. 1995

[9] Hopcroft, J. E., Karp, R. M. An $n^{5/2}$ algorithm for maximum matching in bipartite graphs. *SIAM Journal on Computing*, 2, 4 : 225–231, 1993

[10] Hubbe, P. D., Freuder, E. C. An efficient cross product representation of the constraint satisfaction problem search space. In *Proceedings of the* 10^{th} *National Conference on Artificial Intelligence*, pages 421–427. San Jose, California, USA, 1992

[11] Jeavons, P. G., Cohen, D. A., Cooper, M. C. A substitution operation for constraints. In *Proceedings of the Second International Workshop on Principal and Proctice of Constraint Programming (PPCP94)*, pages 1–9, Lecture Notes in Computer Science 874 Springer-Verlag, 1994

[12] Lal, A., Choueiry, B.Y., Freuder, E.C. Neighborhood interchangeability and dynamic bundling for non-binary finite CSPs. In *Proceedings of AAAI-05*, pages 397–404, 2005

[13] Lawler, E. L., et Wood, D. E. Branch-and-bound methods : A survey. *Operations Research*, 14 :699-719 1966

[14] Lecoutre, C. Toward Benchmarks 2.1. XML representation of CSP instances. *http ://www.cril.univ-artois.fr/∼lecoutre/research/benchmarks/benchmarks.html*, 2007

[15] Lesaint, D. Maximal sets of solutions for constraint satisfaction problems. In *Proceedings of the* 11^{th} *European Conference on Artificial Intelligence*, pages 110–114, Amsterdam, The Netherlands, 1994

[16] Mackworth., A., K. consistancy in networks of relations. *Artificial Intelligence*, 8 : 99–118, 1977

[17] Naanaa, W. Directional interchangeability for enhancing CSP solving. *CPAIOR 2007*, pages 200–213, Hentenryck and Wolsey Eds. LNCS 4510, 2007

[18] Neagu, N., Bistarelli, S., Faltings, B. Experimental Evaluation of Interchangeability in Soft CSPs. *International Workshop on Constraint Solving and Constraint Logic Programming (CSCLP 2003)*, pages 140–153. 2003

[19] Sabin, D., Freuder, E.C. Contradicting conventional wisdom in constraint satisfaction. *Proceedings of the 11th European Conference on Artificial Intelligence*, 1994

[20] Schiex, T., Fargier, H., Verfaillie, G. Valued constraint satisfaction problems : hard and easy problems. In *Proceedings of the* 14^th *IJCAI*, pages 631–637, Montréal, Canada, 1995

[21] Silaghi, M. C., Sam-Haroud, D., Faltings, B. V. Ways of maintaining arc-consistency in search using the Cartesian representation *New Trends in Constraints, Joint ERCIM/Compulog Net Workshop*, pages 173–187, Springer-Verlag, 2000

[22] Taillard, E. Benchmarks for basic scheduling problems. *European Journal of Operational Research*, 64 : 278–285, 1993

[23] Tsang, E. Foundations of constraint satisfaction. *Published by Academic Press Limited in UK* 24-28 Oval Road, London NW1 7DX USA : Sandiego, CA 92101 ISBN 0-12-701610-4. 1993

[24] Zhang, Y., Freuder, E.C. Conditional interchangeability and substitutability. In *Proceedings of Fourth International Workshop on Symmetry and Constraint Satisfaction Problems (SymCon04)*, Toronto, 2004

	Min. conflit				Min. valeur			
	temps		coût		temps		coût	
inst.	BAB	BAB-DS	BAB	BAB-DS	BAB	BAB-DS	BAB	BAB-DS
105-0	7065	651	1308	1308	–	225	–	1290
105-1	90	3847	1326	1325	–	1573	–	1304
105-2	11521	748	1295	1295	–	1173	–	1285
105-3	–	3743	–	1239	–	6193	–	1234
105-4	–	5503	–	1297	4491	12943	1291	1285
105-5	–	2532	–	1308	–	2249	–	1309
105-6	105	6056	1297	1296	2292	9559	1284	1286
105-7	–	376	–	1282	–	523	–	1282
105-8	3406	74	1353	1353	–	2541	–	1343
105-9	3051	4681	1334	1333	739	13850	1334	1316

	Min. conflit				Min. valeur			
	temps		coût		temps		coût	
inst.	BAB	BAB-DS	BAB	BAB-DS	BAB	BAB-DS	BAB	BAB-DS
104-0	–	4526	–	1296	12157	3366	1292	1279
104-1	–	460	–	1313	–	12147	–	1302
104-2	–	88	–	1283	–	13812	–	1269
104-3	646	2716	1228	1227	–	11080	–	1228
104-4	1500	12942	1285	1285	–	10504	–	1281
104-5	–	643	–	1297	–	–	–	–
104-6	–	1136	–	1284	–	766	–	1282
104-7	2606	14028	1270	1270	–	–	–	–
104-8	61	539	1341	1340	–	4137	–	1340
104-9	99	79	1321	1321	739	–	1318	–

	Min. conflit				Min. valeur			
	temps		coût		temps		coût	
inst.	BAB	BAB-DS	BAB	BAB-DS	BAB	BAB-DS	BAB	BAB-DS
103-0	–	122	–	1284	–	–	–	–
103-1	–	885	–	1301	–	3813	–	1284
103-2	–	423	–	1271	–	389	–	1262
103-3	–	–	–	–	–	9734	–	1213
103-4	–	–	–	–	–	218	–	1270
103-5	–	–	–	–	–	–	–	–
103-6	–	1274	–	1273	–	–	–	–
103-7	2513	–	1259	–	–	–	–	–
103-8	–	–	–	–	–	132	–	1327
103-9	–	454	–	1309	–	8926	–	1287

FIGURE 4 – Résultats obtenus par BAB et BAB-DS sur les instances js-taillard-15-(105 à 103) avec deux heuristiques d'ordre des valeurs : min. conflit et min. valeur. Le temps CPU en secondes et le coût de la meilleure solution trouvée sont indiqués. Pour le temps CPU, on indique les temps auxquels ont été trouvées les meilleures solutions. Le temps d'exécution a été limité à 4 heures par instance.

Recherche de la substituabilité par l'arc-cohérence de singleton

Dominique D'Almeida **Lakhdar Saïs**

CRIL-CNRS, Université d'Artois
Rue Jean Souvraz SP18, F-62307 Lens Cedex France

{dalmeida,sais}@cril.fr

Résumé

Dans ce papier, une généralisation sémantique de la substituabilité des valeurs au voisinage est présentée. Notre généralisation est obtenue par une substitution de la notion syntaxique de supports avec une mesure sémantique de la propagation de valeurs obtenues par une utilisation classique de l'arc-cohérence de singleton. Ces nouvelles formes généralisées sont alors exploitées de deux manières différentes. Dans un premier temps, un nouveau prétraitement, intégré à l'algorithme d'Arc-Cohérence de Singleton, des réseaux de contraintes est proposé. Ensuite, puisque l'arc-cohérence de singleton est une opération de base des algorithmes de résolution de type MAC (choix+propagation), nous montrons que notre généralisation peut être facilement exploitée dynamiquement. Les résultats expérimentaux de notre approche se montrent intéressants sur certaines classes d'instances CSP pour le prétraitement, et permettent de montrer que la recherche dynamique de valeurs substituables aux algorithmes de recherche classiques s'avère rentable.

1 Introduction

Le problème de satisfaction de contraintes (CSP) est un cadre général de modélisation et de résolution de problèmes combinatoires, dans notre cas, à domaines finis discrets. Il est défini comme le problème consistant à trouver des valeurs à affecter aux variables de manière à satisfaire des contraintes généralement exprimées par l'ensemble des combinaisons de valeurs autorisées (ou, dualement, interdites). Ce problème de décision NP-Complet, a conduit de nombreux chercheurs à mettre en œuvre des approches algorithmiques pour résoudre aussi efficacement que possible le cas général. Ceci a permis de définir de nouveaux paradigmes de résolution et à en améliorer les étapes, telle que le prétraitement, la propagation de contraintes, le choix de la variable ou le choix de la valeur à affecter. Dans le cas d'une recherche arborescente, il est possible d'entrevoir deux types de stratégies, non exhaustives, qui, a un instant τ de la recherche, visent à comprendre et à analyser ce qui s'est passé avant τ pour mieux anticiper le futur de la recherche.

Pour les stratégies prenant en compte le passé (approches rétrospectives), l'analyse de conflits et la génération de nogoods (ensembles de choix d'affectations incohérents) restent des moyens efficaces d'élaguer un espace de recherche, c'est-à-dire d'éviter d'évaluer un sous-espace de recherche redondant qui mènerait à une incohérence, mais peuvent être coûteuses ou difficiles à mettre en place. Pour les stratégies privilégiant le futur (approches prospectives), il est possible de considérer les techniques de propagations de contraintes, par exemple, qui permettent de supprimer les valeurs qui s'avèrent problèmatiques (incohérentes) dans la suite de la résolution. Mais il y a aussi des traitements efficaces prenant en compte ces deux types de stratégies. L'un des exemples est l'heuristique de choix de variables dom/wdeg qui utilise la pondération de contraintes préalablement falsifiées pour choisir la prochaine variable à affecter.

D'autres types de traitements ont aussi fait l'objet de nombreux travaux, comme la détection et l'exploitation de différentes formes de structures du problème. Parmi elles, nous pouvons citer les symétries de manière générale [1, 6], et des formes plus faibles telles que l'interchangeabilité et la substituabilité de valeurs au voisinage [3]. Ces dernières sont plus simples à détecter (inclusions de tuples autorisés au voisinage) et

à exploiter (suppressions de valeurs) que les symétries dont le problème de la détection, équivalent à l'automorphisme de graphes, est difficile dans le cas général. Ces structures sont détectées, de manière générale, dans leurs formes syntaxiques au niveau du prétraitement et exploitées ensuite de manière statique ou dynamique [2, 4].

Dans ce cadre, l'approche que nous proposons est basée sur une généralisation sémantique de la substituabilité et, parallèlement, de l'interchangeabilité de valeurs au voisinage. Celles-ci sont effectuées en remplaçant la notion syntaxique de supports par une mesure sémantique de la propagation des contraintes obtenues par une utilisation classique de l'arc-cohérence de singleton. Cette généralisation admet plusieurs avantages. Premièrement, elle peut être mise en œuvre de manière statique dans le cadre d'un prétraitement intégré à l'algorithme d'Arc-Cohérence de Singleton (SAC) en donnant lieu à une double généralisation de cet algorithme et de la substituabilité au voisinage. Ce prétraitement vise à faciliter la détection de valeurs substituables sans surcoût pendant la recherche. Passer du temps pour le prétraitement peut avoir des conséquences avantageuses pour la suite de la résolution, et le filtrage par SAC en est un exemple. Le second avantage réside dans la simplicité de mise en œuvre dynamique. En effet, comme la vérification de la cohérence d'arc des singletons est une opération de base des algorithmes de type MAC, il est possible d'exploiter cette généralisation de manière dynamique au moment de l'affectation d'une valeur d'une variable. Celle-ci combine donc les stratégies "passé" et "futur", et permet d'agir en cours de recherche. Elle permet de conserver une trace de l'état du réseau obtenu par la propagation des contraintes après affectation d'une variable, pour vérifier par la suite si cet état réapparaît et éviter ainsi une exploration redondante.

Après avoir défini quelques notions de bases essentielles à la compréhension du problème, nous décrirons notre généralisation de la substituabilité au voisinage par l'arc-cohérence de singleton (3.1), en étudiant les approches statiques (3.2) et dynamiques du problème (3.3). La section 4 présentera la validation expérimentale de ces approches avant de conlure en énonçant quelques perspectives.

2 Notion de bases

Un réseau de contraintes $\mathcal{P}$ est un couple $(\mathcal{X}, \mathcal{C})$ avec $\mathcal{X} = \{x_1, ..., x_n\}$ un ensemble de variables et $\mathcal{C} = \{c_1, ..., c_e\}$ un ensemble de contraintes entre les variables de $\mathcal{X}$. À chaque variable x_i de $\mathcal{X}$ est associé un ensemble de définition fini $dom(x_i) = \{v_1, ..., v_d\}$, appelé domaine de x_i. Une affectation (resp. réfutation) est un couple $(x, v) \in (\mathcal{X}, dom(x))$, noté $x = v$ (resp. $x \neq v$), tel que $dom(x) = \{v\}$ (resp. v est supprimée de $dom(x)$).

Chaque contrainte c_i est définie par un couple (s_i, r_i) tel que :

- $s_i \subseteq \mathcal{X}$ est appelé portée de la contrainte c_i et correspond aux variables sur lesquelles s'applique la contrainte c_i. On note $card(s_i) = a_i$ l'arité de la contrainte c_i ;
- $r_i \subseteq \prod_{x \in s_i} dom(x)$ est appelée relation et représente un ensemble de tuples autorisés, appelés supports de c_i, entre les variables de s_i.

Relativement à la portée, deux variables distinctes x et y sont dites voisines ssi il existe une contrainte $c_i \in \mathcal{C}$, appelée contrainte voisine de x et y, telle que $\{x, y\} \subseteq s_i$. Un ensemble de variables $X \subseteq \mathcal{X}$ (resp. ensemble de contraintes $C \subseteq \mathcal{C}$) est au voisinage d'une variable x ($x \notin X$) si et seulement si pour tout $x_i \in X$ (resp. $c_i \in C$), x est voisine de x_i (resp. c_i).

En considérant la notion de tuple, nous noterons $dom(c_i)$ le domaine de définition de c_i tel que $dom(c_i) = r_i$. Chaque tuple $t \in dom(c_i)$ est défini par un $a_i - uplet$ $(v_1, ..., v_{a_i})$. Nous utiliserons la notation $t[x_j]$ pour désigner la valeur v_j associée à x_j et $t[\hat{x_j}]$ pour représenter $(v_1, ..., v_{j-1}, v_{j+1}, ..., v_{a_i})$. Un tuple $t \in dom(c_i)$ est un support pour $x = v$ ($x \in s_i$ et $v \in dom(x)$) si et seulement si $t[x] = v$. L'ensemble des supports pour $x = v$ dans c_i est noté $supports(c_i|_{x=v})$. Une contrainte est satisfaite ssi elle contient au moins un support.

Une solution pour un réseau de contraintes $\mathcal{P} = (\mathcal{X}, \mathcal{C})$ est une affectation de chaque variable de $\mathcal{X}$ satisfaisant toutes les contraintes. Un réseaux de contraintes est dit cohérent (ou satisfaisable) s'il admet au moins une solution, et incohérent (ou insatisfaisable) sinon. Le problème de satisfaction de contraintes CSP est le problème de décision (NP-complet) qui consiste à déterminer si un réseau de contraintes admet ou non une solution. Nous évoquerons la résolution d'une instance CSP, définie par un réseau de contraintes $\mathcal{P}$, comme étant la recherche d'une solution ou la démonstration de l'insatisfaisabilité.

Pour la résolution de CSP, nous considérons uniquement des algorithmes de recherche en profondeur d'abord, de type retours-arrière (Backtrack ou BT), maintenant la cohérence d'arc à chaque noeud de l'arbre (MAC pour Maintain Arc Consistency). Cet algorithme de type choix+propagation est complet, c'est-à-dire qu'il amène nécessairement à une réponse en parcourant, dans le pire des cas, l'ensemble de l'espace de recherche.

La Cohérence d'Arc est un processus de filtrage appelé propagation de contraintes. Dans un CSP une valeur v de $dom(x)$ est arc-cohérente pour une contrainte

c_i ($x \in s_i$) ssi il existe un support pour $x = v$ dans c_i. Une variable est arc-cohérente pour une contrainte ssi toutes ses valeurs sont arc-cohérentes. Une variable est arc-cohérente ssi elle est arc-cohérente pour toutes les contraintes dans lesquelles cette variable est impliquée. Un réseau est arc-cohérent ssi toutes ses variables sont arc-cohérentes. Sans distinction sur l'arité des contraintes du réseau, la Cohérence d'Arc est appelée Cohérence d'Arc Généralisée (GAC) D'autres types de cohérences existent et permettent un filtrage, plus ou moins fort, des variables en fonction des contraintes du réseau. On définit $\phi(\mathcal{P})$ le réseau de contraintes obtenu après l'application de l'algorithme de propagation de contraintes ϕ. Par exemple, ϕ = GAC signifie que toutes les valeurs arc-incohérentes de $\mathcal{P}$ sont supprimées. Dans la suite nous noterons, pour simplifier, le réseau de contraintes obtenu par l'application de la Cohérence d'Arc AC($\mathcal{P}$). S'il existe une variable ayant un domaine vide dans $\phi(\mathcal{P})$, on note $\phi(\mathcal{P}) \models \bot$ l'incohérence résultant de ϕ. Le sous-réseau obtenu après l'affectation $x = v$ est noté $\mathcal{P}|_{x=v}$.

La résolution de type MAC impose de spécifier le type de branchement utilisé pour développer l'arbre de recherche. En effet, il est possible de considérer un branchement binaire (2way-branching) ou non binaire (dway-branching). Par 2way-branching , on entend qu'à chaque étape un couple variable-valeur (x, v) est sélectionné et deux cas sont ensuite considérés : l'affectation $x = v$ et, si celle-ci n'aboutit pas à une solution, la réfutation $x \neq v$ qui se poursuit du choix d'un nouveau couple. Pour le $d$way-branching , à chaque étape, une variable x est sélectionnée et affectée à une valeur v de $dom(x)$. Si cette valeur est réfutée, une autre valeur de $dom(x)$ est choisie puis affectée, ceci tant que la réfutation n'amène pas à une incohérence locale. Chacun de ces modèles a des avantages, le 2way-branching est néanmoins préféré en pratique puisqu'il donne une importance majeure à l'heuristique de choix de variables.

En suivant ces modèles axés sur le développement d'un arbre de recherche, nous pouvons définir une notion d'instanciation pour un CSP $\mathcal{P} = (\mathcal{X}, \mathcal{C})$. L'instanciation des variables de $X \subseteq \mathcal{X}$ est une application $\mathcal{I}_X$ de X dans $dom(x)$ qui a x associe v, décrivant l'affectation $x = v$. $\mathcal{I}_X$ est appelée instanciation complète si $X = \mathcal{X}$ et instanciation partielle sinon. $\mathcal{I}_\emptyset$ représentera l'absence d'affectation. Une instanciation $\mathcal{I}_X$ est cohérente si et seulement si elle vérifie les contraintes du réseau. Une instanciation complète et cohérente est une solution.

Ces quelques notions permettent de définir notre cadre, dont le travail préliminaire passe par la définition et l'étude de la substituabilité au voisinage, ce qui permettra une meilleure compréhension de l'intérêt d'intégration de cette approche aux algorithmes existant.

3 Substituabilité basée sur l'arc-cohérence de singleton

3.1 Définition et étude de la substituabilité au voisinage

La substituabilité et, parallèlement, l'interchangeabilité sont les maîtres-mots de notre contribution. Ces propriétés des CSP ont donné lieu à de nombreux travaux mettant en évidence la difficulté à les déterminer [3]. Il existe deux formes de substituabilité et interchangeabilité : complète ou local.

Pour un CSP $\mathcal{P} = (\mathcal{X}, \mathcal{C})$ et deux couples distincts $(x, v), (x, v') \in (\mathcal{X}, dom(x))$, v est complètement substituable par v' ssi pour chaque solution $\mathcal{I}_\mathcal{X}$ telle que $\mathcal{I}_\mathcal{X}(x) = v$, il existe une solution $\mathcal{I}'_\mathcal{X}$ telle que $\mathcal{I}'_\mathcal{X}(x) = v'$ et $\mathcal{I}'_\mathcal{X}(y) = \mathcal{I}_\mathcal{X}(y)$ pour tout $y \in \mathcal{X}$ et $y \neq x$. Il est facile de voir que les notions d'interchangeabilité complète et de substituabilité complète sont liées. En effet, deux valeurs v et v' sont complètement interchangeables ssi v est complètement substituable par v' et v' est complètement substituable par v. Deux valeurs complètement interchangeables sont donc complètement substituables. Cette définition montre la difficulté à déterminer l'interchangeabilité ou la substituabilité complète de deux valeurs, dont la complexité est coNP-complet. Cela nécessite l'énumération des solutions (variable de référence exclue) et la vérification de l'inclusion de chacune dans les autres.

Cependant, un affaiblissement permet de définir une forme locale de la substituabilité et de l'interchangeabilité de deux valeurs : substituabilité au voisinage et interchangeabilité au voisinage. Celles-ci se définissent par la substituabilité ou l'interchangeabilité entre deux valeurs d'une variable x sur un sous-ensemble de variables ou de contraintes du réseau, notamment au voisinage de x. Dans la suite, nous nous concentrerons sur la substituabilité au voisinage, que nous privilégierons à l'étude de l'interchangeabilité car, comme nous l'évoquerons, déterminer la substituabilité suffit dans notre cadre. De façon analogue à la forme de substituabilité complète, il nous est possible de définir la substituabilité au voisinage comme suit :

Définition 1 *Soit $\mathcal{P} = (\mathcal{X}, \mathcal{C})$ un CSP, soient deux couples distincts $(x, v), (x, v') \in (\mathcal{X}, dom(x))$ et X l'ensemble de variables composé de x et de ses voisines. v est substituable par v' au voisinage de x si et seulement si, pour chaque instanciation partielle cohérente $\mathcal{I}_X$ telle que $\mathcal{I}_X(x) = v$, il existe une instanciation partielle cohérente $\mathcal{I}'_X$ telle que $\mathcal{I}'_X(x) = v'$ et $\mathcal{I}'_X(y) = \mathcal{I}_X(y)$, pour tout $y \in X$ et $y \neq x$.*

Cette définition se concentre cette fois sur l'ensemble de variables X, correspondant à la variable x et son voisinage, pour mettre en évidence la relation entre l'état du voisinage de x pour l'une et l'autre des valeurs de son domaine. Dans la littérature, il est possible de trouver une reformulation plus classique de cette définition [3] qui se concentre sur la notion de supports au voisinage :

Définition 2 *Pour un* CSP $\mathcal{P} = (\mathcal{X}, \mathcal{C})$ *et deux couples* $(x, v), (x, v') \in (\mathcal{X}, dom(x))$. *$v$ est substituable par v' au voisinage de x ssi, pour toute contrainte c_i voisine de x, $supports(c_i|_{x=v}) \subseteq supports(c_i|_{x=v'})$.*

Intuitivement, la notion de supports peut être associée à la notion de propagation de contraintes pour énoncer les propositions suivantes :

Proposition 1 *Soit $\mathcal{P} = (\mathcal{X}, \mathcal{C})$ un* CSP*, soient deux couples $(x, v), (x, v') \in (\mathcal{X}, dom(x))$, et ϕ un algorithme de propagation de contraintes. v est substituable par v' au voisinage de x si et seulement si le* CSP *$\phi(\mathcal{P}|_{x=v})$ est un sous-réseau de $\phi(\mathcal{P}|_{x=v'})$, à l'exception de la variable x.*

Proposition 2 *Soit $\mathcal{P} = (\mathcal{X}, \mathcal{C})$ un* CSP*, soient deux couples $(x, v), (x, v') \in (\mathcal{X}, dom(x))$ tels que v est substituable par v' au voisinage de x , et ϕ un algorithme de propagation de contraintes. Si $\phi(\mathcal{P}|_{x=v'}) \models \bot$ alors $\phi(\mathcal{P}|_{x=v}) \models \bot$.*

Ces deux résultats forment le centre d'intérêt de notre contribution. Pour prouver ces propositions, il suffit de se concentrer sur les définitions 1 et 2. Le sens "si" est immédiat. Puisque v est substituable par v' au voisinage de x, nous avons, d'après la définition 2, $supports(c_i|_{x=v}) \subseteq supports(c_i|_{x=v'})$. En itérant la propagation des contraintes jusqu'à atteindre un point fixe du CSP, les supports $supports(c|_{x=v})$ de $\phi(\mathcal{P}|_{x=v})$ sont inclus dans les supports $supports(c|_{x=v'})$ pour tout c, à l'exception de la variable x . Donc CSP $\phi(\mathcal{P}|_{x=v})$ est un sous-réseau de $\phi(\mathcal{P}|_{x=v'})$, variable x exclue.

Dans l'autre sens, nous avons $\phi(\mathcal{P}|_{x=v})$ sous-réseau de $\phi(\mathcal{P}|_{x=v'})$, en omettant la variable x, et ϕ un algorithme vérifiant la cohérence locale du réseau. Donc pour chaque instanciation localement cohérente $\mathcal{I}_{X\setminus\{x\}}$ (avec X ensemble de variable composé de x et de son voisinage) de $\phi(\mathcal{P}|_{x=v})$, il existe une instanciation localement cohérente $\mathcal{I}'_{X\setminus\{x\}}$ telle que $\mathcal{I}'_{X\setminus\{x\}}(y) = \mathcal{I}_{X\setminus\{x\}}(y)$, pour tout $y \in X$ et $y \neq x$. En considérant la variable x dans chacune des instanciations, nous obtenons $\mathcal{I}_X(x) = v$ et $\mathcal{I}'_X(x) = v'$, mais aussi $\mathcal{I}'_{X\setminus\{x\}}(y) = \mathcal{I}_{X\setminus\{x\}}(y)$. D'après la définition 1, v est donc substituable par v' au voisinage de x. Enfin, trivialement, en considérant la proposition 1, chaque solution, variable x exclue, du CSP $\phi(\mathcal{P}|_{x=v})$ l'est aussi pour $\phi(\mathcal{P}|_{x=v'})$. Par contraposée, si le CSP $\phi(\mathcal{P}|_{x=v'})$ est incohérent, alors $\phi(\mathcal{P}|_{x=v})$ l'est aussi. Ce qui montre la proposition 2.

L'idée est donc d'utiliser et d'intégrer le plus efficacement possible ces résultats aux algorithmes de prétraitement et de recherche comme, par exemple, SAC et MAC. Avant d'exploiter les valeurs substituables au voisinage, il faut néanmoins remplir cette condition de substituabilité. Pour cela, il suffit de se concentrer sur la définition 2 et de vérifier l'inclusion des supports des contraintes voisines pour les affectations $x = v$ et $x = v'$. Notons que l'utilisation de tuples autorisés, pour le calcul de l'inclusion, est privilégiée dans nos descriptions mais est, dans le cas général, équivalente pour la vérification d'inclusion à l'utilisation des tuples interdits. Aussi, une remarque peut être faite quant à l'efficacité, en nombre de valeurs substituables, de la détection de la substituabilité au voisinage, puisque celle-ci est dépendante de l'algorithme de propagation de contraintes. Dans la suite, nous concentrerons notre étude sur la propagation par Cohérence d'Arc, notamment utilisé dans les algorithmes SAC et BT+MAC.

3.2 Substituabilité au prétraitement SNS

Pour correspondre au mieux à notre étude de la substituabilité au voisinage, le choix de l'intégration au prétraitement se porte sur l'algorithme SAC. Pour un CSP $\mathcal{P} = (\mathcal{X}, \mathcal{C})$ et une variable $x \in \mathcal{X}$, une valeur $v \in dom(x)$ est singleton arc-cohérente (sac) ssi $AC(\mathcal{P}|_{x=v}) \not\models \bot$. Et, de manière étendue, une variable x est sac ssi, pour tout $v \in dom(x)$, v est sac. L'Arc-Cohérence de Singleton (SAC) est une forme plus forte de cohérence locale que GAC qui garantit $\phi(\mathcal{P}|_{x=v}) \not\models \bot$, pour chaque couple variable-valeur (x, v). La propagation suivant l'affectation de chaque couple (x, v) présente un grand intérêt dans notre cas puisque nous pourrons profiter du traitement efficace effectué par cet algorithme et y greffer notre approche, tout en bénéficiant du filtrage effectué par SAC.

L'algorithme résultant de l'intégration de notre approche, que nous nommons SNS pour "*SAC and Neighbouhood Substituability*", se définit en deux étapes. La première se concentre sur la détection des valeurs substituables. Pour cela, les supports des contraintes voisines sont stockés après l'appel à l'algorithme AC sur le réseau $\mathcal{P}|_{x=v}$, dans une structure correspondant à un état du voisinage de x (noté $\mathcal{E}_{x=v}$). L'inclusion des états s'effectue ensuite simplement en vérifiant l'inclusion des supports de chaque contrainte de l'un et l'autre des états. Formellement, en considérant $\mathcal{E}_{x=v}(c)$ la contrainte c dans $\mathcal{E}_{x=v}$, nous avons $\mathcal{E}_{x=v} \subseteq \mathcal{E}_{x=v'}$ ssi, pour tout tuple $t \in \mathcal{E}_{x=v}(c)$, il existe un tuple

$t' \in \mathcal{E}_{x=v'}$ tel que $t[\hat{x}] = t'[\hat{x}]$. S'il y a inclusion pour toutes les contraintes, il y a inclusion pour l'état. La seconde étape utilise la proposition 1. Dans le cadre de SNS, si v est substituable par v' au voisinage de x, les solutions obtenues avec l'affectation $x = v$ permettent de déduire certaines solutions pour $x = v'$, donc v est supprimée. La détection de l'interchangeabilité au voisinage n'apporte donc pas de nouvelles informations puisque l'une ou l'autre des variables est destinée à être supprimée.

Associée à SAC, cette approche, présentée dans l'algorithme 1, permet donc un filtrage des valeurs "non sac" et substituables au voisinage, ce qui le rend plus fort que l'algorithme SAC. Dans celui-ci, seules les instructions 7 à 11 se concentrent sur la recherche de valeurs substituables, le reste de l'algorithme correspond à SAC, ce qui montre la facilité d'intégration.

Algorithme 1 : SNS

```
   Entrées : P = (X, C) : un CSP
   Sorties : SNS(P)
1  début
2      répéter
3          pour chaque couple (x, v) ∈ (X, dom(x)) faire
4              si AC(P|_{x=v}) ⊨⊥ alors
5                  Supprimer (x, v)
6              sinon
7                  Stocker E_{x=v}
8                  pour chaque (x, v') prouvé sac faire
9                      E_{x=v} ⊆ E_{x=v'} ⇒ Supprimer (x, v)
10                     Et si v n'est pas supprimé :
11                     E_{x=v'} ⊆ E_{x=v} ⇒ Supprimer (x, v')
12                 fin
13             fin
14         fin
15     jusqu'à ne plus avoir de suppression de couples
16 fin
```

Aussi, l'efficacité en temps et en espace d'un tel calcul d'inclusion des supports dépend directement de l'arité de la contrainte. Si la contrainte c implique a variables ayant chacune un domaine de taille d, la contrainte peut avoir, dans le pire des cas, d^a supports.

Avant d'entamer le calcul de la complexité algorithmique de SNS, nous rappelons que le réseau de contraintes possède n variables de domaine d, que le nombre de contraintes est e et chacune est d'arité maximale r.

Pour la complexité en espace, considérons tout d'abord l'espace nécessité par un état pour un couple variable-valeur (x, v). Puisque l'arité des contraintes est r, chaque variable est voisine de $r-1$ variables, dont le domaine est d. Il y a d^{r-1} supports par contrainte et donc $(r-1)d^{r-1}$ valeurs par contrainte. Dans le pire des cas, chaque variable est impliquée dans les e contraintes du réseau, ce qui permet de déduire qu'il y a $e(r-1)d^{r-1}$ valeurs par état, dans le pire des cas. Pour comparer les états correspondant à chaque valeur d'une variable, SNS nécessite de conserver d états. Nous avons donc une complexité en espace en $O(erd^r)$ pour l'ensemble des états à stocker lors de SNS.

Pour calculer la complexité en temps de SNS, il faut tout d'abord considérer que celui-ci est intégré à l'algorithme SAC, puis décomposer SNS. Tout d'abord, il est possible de noter qu'un traitement est effectué pour chaque couple variable-valeur, soit nd fois. Ce traitement correspond à la cohérence d'arc (1), le stockage de l'état associé au couple (2) et la comparaison avec les états existant pour une même variable (3). La complexité de l'algorithme AC est en $O(erd^r)$ (1). Le stockage d'un état nécessite le parcours de ses $e(r-1)d^{r-1}$ valeurs (2). Chacun de ces états doit être comparé aux états précédemment conservés. L'inclusion sera testée dans un sens puis dans l'autre, ce qui nécessite $d-1$ vérifications d'inclusion pour chaque couple variable-valeur. L'inclusion de deux états implique le parcours de l'ensemble des valeurs de ces états, donc erd^r valeurs pour les vérifications d'inclusion (3). En détail, nous avons donc une complexité en temps pour le traitement énoncé précédemment proche de $nd \times (erd^r + e(r-1)d^{r-1} + erd^r)$, soit $O(ernd^{r+1})$. Or, comme dans le cadre de l'algorithme SAC classique, la suppression d'une valeur dans le réseau de contraintes, quelle que soit la raison, implique la réitération du traitement effectué ci-dessus puisque l'état du réseau à changer. Comme le réseau possède nd valeurs, le traitement doit, dans le pire des cas, être itéré nd fois. La complexité en temps, permettant de détecter l'ensemble des valeurs non-sac et les valeurs substituables au voisinage est $O(ern^2d^{r+2})$.

Bien qu'ayant des complexités en espace et en temps élevées à cause de l'arité des contraintes, il faut néanmoins noter qu'en pratique, l'efficacité de l'approche dépendra, bien évidemment, des traitements et des structures utilisées. Aussi, puisque cette complexité est fonction de l'arité, il est intéressant de voir le cas particulier des contraintes binaires, qui peut être considéré comme le cas le plus "léger" pour SNS. La complexité en espace est en $O(end^2)$ et la complexité en temps est $O(en^2d^4)$, c'est-à-dire comparable à l'algorithme SAC classique dans le pire des cas.

La détection de valeurs substituables au voisinage dans le cadre d'un prétraitement peut donc s'avérer intéressant notamment par son intégration facile dans des algorithmes tel que SAC, permettant d'approfondir leurs capacités. Pourtant, les pleines ressources de notre prétraitement ne sont pas développées ici mais sont nombreuses. Il est de notre avis qu'il est possible de déduire des informations sur le réseau, par exemple structurelles, par notre prétraitement pour faciliter l'utilisation d'une approche dynamique.

3.3 Recherche dynamique de la substituabilité

Le développement des différentes étapes de l'approche statique, effectué en 3.2, illustre tous les traitements que nous allons étendre au cas du 2way-branching, permettant de déduire la substituabilité au voisinage dynamiquement. Tout d'abord, l'aspect dynamique de cette intégration vient de la modification constante du réseau de contraintes. Dans le cas d'un algorithme de type BT+MAC, le maintien de l'arc-cohérence dans le réseau se fait quelle que soit la décision qui a été prise, notamment après affectation d'une variable. La détection de valeurs substituables au voisinage d'une variable pourra donc se greffer facilement, comme dans le cadre statique, et être calculée à chaque nœud de l'arbre de recherche.

De manière analogue à l'approche SNS, les supports au voisinage sont stockés après chaque propagation des contraintes suivant une affectation $x = v$. Les états d'une variable sont donc décrits par l'état produit lors de la dernière affectation (l'état courant) et par les états conservés pour les valeurs précédemment réfutées (les états passés). D'après la proposition 2, si l'état courant $\mathcal{E}_{x=v}$ est inclus dans un état passé $\mathcal{E}_{x=v'}$, alors la valeur v est substituable par v' et puisque v' a été réfutée, v doit l'être elle aussi. Notons que ce résultat est dû à la réfutation du couple (x, v') à la racine de l'arbre de recherche, ce qui induit une incohérence globale de celui-ci. Le choix $x = v$, menant à un sous-état de celui produit par $x = v'$, produit nécessairement une incohérence globale de (x, v). L'approche dynamique permet de ne pas à avoir à réévaluer certains sous-espaces de recherche localement cohérents mais globalement incohérents pouvant être rencontrés s'il n'y a pas détection de la substituabilité des valeurs au voisinage d'une variable. Pour cette approche, la détection de l'interchangeabilité au voisinage est donc impossible puisque la vérification d'inclusion est unilatérale et imposée par la recherche. Les lignes 5 à 7 de l'algorithme ns+dual (Algo. 2) illustrent la description effectuée ci-dessus, intégrée à l'algorithme classique 2way-branching de type BT+MAC.

L'intégration de la détection de la substituabilité au 2way-branching peut, bien entendu, être adaptée au cas du $d$way-branching (que nous appelerons ns+dway) de manière similaire puisque les différences entre ces algorithmes sont indépendantes de notre approche. Cependant, malgré la facilité de mise en œuvre, la condition de la ligne 13 de l'algorithme ns+dual souligne un fait important dans notre cadre. La comparaison entre les états doit être faite sous certaines conditions, notamment liées à l'état du réseau avant l'affectation. Par exemple, comparer un état au voisinage de x pour le réseau $\phi(\mathcal{P}|_{y=0,x=0})$ à un état au voisinage de x pour le réseau $\phi(\mathcal{P}|_{y=1,x=1})$ n'a pas de sens dans le cas de la détection de la substituabilité au voisinage, à moins que les réseaux $\phi(\mathcal{P}|_{y=0})$ et $\phi(\mathcal{P}|_{y=1})$ soient équivalents.

Algorithme 2 : ns+dual

```
   Entrées : P = (X, C) : un CSP
   Sorties : ⊤ si P est cohérent, ⊥ sinon
1  début
2      tant que il existe une variable non affectée faire
3          Choisir (x, v) ∈ (X, dom(x)) tel que x non affectée
4          Affecter v à x et propager les contraintes
5          Stocker l'état du voisinage E_{x=v}
6          pour chaque état E_{x=v'} conservé faire
7              E_{x=v} ⊆ E_{x=v'} ⇒ P ⊨ ⊥
8          fin
9          tant que P ⊨ ⊥ et ∃ une variable affectée faire
10             Restaurer le niveau précédent
11             Réfuter le dernier couple (x', v') et Propager
12             si P ⊨ ⊥ alors
13                 Supprimer les états inutiles
14             fin
15         fin
16         si P ⊨ ⊥ et ∄ une variable affectée alors
17             retourner ⊥
18         fin
19     fin
20     retourner ⊤
21 fin
```

Sans nous concentrer sur les cas particuliers, un critère nécessaire à la comparaison de deux états relève des instanciations desquelles ils sont produits. Par exemple, un état obtenu à la racine de l'arbre de recherche peut-il être comparé à des états produits après un nombre quelconque d'affectations ? La réponse s'appuie sur l'utilisation de la proposition 1 :

Proposition 3 *Soit $\mathcal{P} = (\mathcal{X}, \mathcal{C})$ un* CSP*, soient deux couples $(x, v), (x, v') \in (\mathcal{X}, dom(x))$, et ϕ un algorithme de propagation de contraintes. Soit $\mathcal{I}_X$ une instanciation cohérente telle que $X \subseteq \mathcal{X} \setminus \{x\}$. v est substituable par v' au voisinage de x si et seulement si le* CSP *$\phi(\mathcal{P}|_{\mathcal{I}_X, x=v})$ est un sous-réseau de $\phi(\mathcal{P}|_{x=v'})$, à l'exception de la variable x.*

En utilisant la proposition 1, la preuve est immédiate. Si v est substituable par v' au voisinage de x, alors $\phi(\mathcal{P}|_{x=v})$ est un sous-réseau de $\phi(\mathcal{P}|_{x=v'})$, variable x exclue. Quelle que soit l'instanciation $\mathcal{I}_X$ avec $X \subseteq \mathcal{X} \setminus \{x\}$, $\phi(\mathcal{P}|_{\mathcal{I}_X, x=v})$ est un sous-réseau de $\phi(\mathcal{P}|_{x=v})$. Donc $\phi(\mathcal{P}|_{\mathcal{I}_X, x=v})$ est un sous-réseau de $\phi(\mathcal{P}|_{x=v'})$, en exluant la variable x. Dans l'autre sens, si $\phi(\mathcal{P}|_{\mathcal{I}_X, x=v})$ est un sous-réseau de $\phi(\mathcal{P}|_{x=v'})$ alors les supports de $\phi(\mathcal{P}|_{\mathcal{I}_X, x=v})$ sont inclus dans les supports de $\phi(\mathcal{P}|_{x=v'})$, en particulier au voisinage de x. Donc d'après la définition 2, v est substituable par v' au voisinage de x. Ce qui prouve la proposition 3.

Notons néanmoins que ce résultat tient uniquement si la vérification de l'inclusion se fait d'un état issu

d'une instanciation dans un état issu de la racine de l'arbre de recherche. En effet, la réfutation du couple (x, v) à la racine de l'arbre induit une incohérence globale de celui-ci, donc les valeurs sustituables par v au voisinage de x sont donc incohérentes localement, relativement à l'état du voisinage, qui est le résultat d'une l'instanciation. Or, dans l'autre sens, l'incohérence résultant d'une instanciation ne permet pas de déduire, dans le cas général, une incohérence globale, et donc ne permet pas de déduire, toujours dans le cas général, la suppression de valeurs affectées à la racine de l'arbre.

De plus, ce résultat peut être élargi aux réseaux $\phi(\mathcal{P}|_{\mathcal{I}_X, x=v})$ et $\phi(\mathcal{P}|_{\mathcal{I}_Y, x=v'})$, pour deux instanciations $\mathcal{I}_X$ et $\mathcal{I}_Y$ avec $Y \subseteq X$. Si l'on considère $\mathcal{P}' = \phi(\mathcal{P}|_{\mathcal{I}_Y})$, nous retrouvons la proposition 3 pour les réseaux $\phi(\mathcal{P}'|_{\mathcal{I}_{X\setminus Y}, x=v})$ et $\phi(\mathcal{P}'|_{x=v'})$. La remarque précédente visant le sens d'inclusion démontre donc pourquoi la condition $Y \subset X$ est imposée pour l'instanciation.

La comparaison entre les états du voisinage pour une même variable est donc réduite, dans notre cadre d'étude, à la comparaison de l'état courant issu d'une instanciation et des états passés issus d'une restriction de cette instanciation, exception faite de la variable pour laquelle les valeurs substituables sont recherchées. Ce critère de comparaison est d'une grande utilité dans le cas du 2way-branching, puisqu'il permet une détection de valeurs substituables au voisinage malgré les affectations effectuées à différents niveaux de la recherche. Ceci s'applique aussi dans le cadre du $d$way-branching, mais la particularité dûe au choix de la variable à affecter permettra aux états d'être conservés pour une même instanciation, jusqu'à la réfutation complète du domaine de la variable.

Avant d'étudier la complexité algorithmique de l'approche dynamique, nous rappelons que nous étudions un réseau de contraintes ayant n variables, de domaines de taille d, chacune impliquée dans e d'arité maximale r. Dans le cas présent, l'étude de la complexité pour l'adaptation ns+dual ou ns+dway est équivalente, nous ne faisons donc pas d'étude spécifique à chacune de ces approches.

Pour la complexité en espace, le pire des cas correspond à la réfutation de d valeurs par variable, qui ont donc nécéssité le stockage de d états. Les n variables imposent donc de consever nd états pour l'ensemble de l'évaluation dans l'arbre de recherche, dans le pire des cas. Puisque chaque état est composé de $e(r-1)d^{r-1}$ valeurs, la complexité en espace dans le pire des cas est $O(ernd^r)$, correspondant au nombre de valeurs à conserver pendant la recherche.

Calculer la complexité en temps dans le pire des cas dépend de la façon de traduire le pire des cas issu du développement d'un arbre de taille exponentiel. Notre choix se porte sur l'étude de la complexité en temps correspondant à la détection de la substituabilité des valeurs d'une variable avant que celle-ci ne soit réfutée. Cette réfutation a lieu lorsque d valeurs ont été réfutées et chaque i^{me} affectation conduit à vérifier les états des $i-1$ valeurs réfutées précédemment. Il y a donc, dans le pire des cas, $\binom{d}{2}$ comparaisons d'états. Puisque chaque comparaison nécéssite le parcours de $e(r-1)d^{r-1}$ valeurs, la complexité en temps dans le pire des case est $O(erd^{r+1})$ pour comparer l'ensemble des états avant réfutation complète de la variable.

Pour fournir un ordre d'idée sur le cas binaire, la complexité en espace dans le pire cas est $O(end^2)$ et la complexité en temps dans le pire des cas est $O(ed^3)$ pour déterminer s'il existe des valeurs substituables avant réfutation complète de la variable.

Après l'étude théorique de notre approche, nous allons déterminer si celle-ci s'avère intéressante en pratique.

4 Expérimentations

Avant de développer les différents cadres étudiés précédemment, quelques petites précisions sur le protocole expérimental. Celui-ci se déroule en 2 phases : approche statique intégrée à SAC avec méthode de recherche 2way-branching classique (appelée SNS), et approche dynamique intégrée à ns+dual avec prétraitement par cohérence d'arc (appelée ns+dual). Chaque approche a été comparée à son homologue dit "de base", c'est-à-dire sans détection de substituabilité au voisinage (respectivement SAC et dual). Les expérimentations ont été réalisées dans le cadre binaire essentiellement. Le cas des instances n-aires fait toujours l'objet d'étude pour la mise en pratique, et les complexités en espace et en temps expliquent ce choix. Certaines approches auraient pu être exploitées, comme imposer une limite à l'arité des contraintes évaluées, mais puisque ceci aurait été complètement arbitraire par manque d'études, nous avons décidé de ne présenter que le cas binaire. L'implémentation a été réalisée en JAVA à l'aide du solveur Abscon109 [5] développé au CRIL, et les expérimentations ont été menées sur un processeur Xéon 3GHz avec 1Go de RAM. Les instances CSP utilisées sont celles de la 3^{eme} compétition internationale de solveurs CSP réalisée en 2008 (http ://cpai.ucc.ie/08/). Ces instances sont normalisées, c'est-à-dire que deux contraintes ne peuvent pas impliquer le même ensemble de variables. Les instances sont divisées en 4 catégories contraintes en extension/intension et SAT/UNSAT. Le temps de résolution a été limité à $1200s$ et la mémoire à 900Mo.

4.1 Intégration statique

La comparaison dans le cadre statique est basée sur les critères suivants : le nombre d'instances résolues, le nombre de valeurs filtrées, les temps de résolution/prétraitement et le nombre de nœuds parcourus.

Extension SAT				
Solveur	#	Temps	# inst.	Filtr.
SAC	296	20624	57	4373
SNS	296	21019	72	41909
Extension UNSAT				
Solveur	#	Temps	# inst.	Filtr.
SAC	195	13824	68	6869
SNS	195	13807	78	8372
Intension SAT				
Solveur	#	Temps	# inst.	Filtr.
SAC	253	14867	57	16404
SNS	254	18054	110	82432
−SNS	253	17855	109	79618
Intension UNSAT				
Solveur	#	Temps	# inst.	Filtr.
SAC	202	8307	47	24382
SNS	205	10857	51	41546
−SNS	202	8017	49	37690

TAB. 1 – Résultats globaux : SAC contre SNS

La tableau 1 présente les résultats généraux de la résolution de toutes les instances binaires. Dans celle-ci, chaque tableau représente une catégorie, pour lesquelles le nombre d'instances (#), le temps de résolution total en seconde (Temps), le nombre d'instances sur lesquelles au moins une valeur a été supprimée (# inst.) et le nombre de valeurs filtrées (Filtr.). Nous considérons le nombre de valeurs filtrées comme étant le nombre de valeurs supprimées au prétraitement, c'est-à-dire les valeurs non-sac et les valeurs substituables dans le cas de SNS. Aussi, le nom d'une approche précédée d'un signe − indique que seules les instances résolues par les deux approches sont prises en compte dans les résultats de celle-ci.

Premier résultat : l'approche basée sur SNS est meilleure en nombre d'instances résolues que l'approche SAC. Elle répond aux problèmes pour 4 instances de plus que l'approche classique. Or SNS n'a joué un rôle que pour 3 d'entre elles. Nous pouvons donc déduire que les caractéristiques spécifiques de l'approche SNS ont permis de conclure pour 3 problèmes de plus que SAC. Un résultat tout aussi intéressant, toutes catégories confondues, est que le nombre de valeurs filtrées par SAC est de 59771, alors que SNS filtre 182041 valeurs, ce qui montrent la puissance de filtrage de notre approche. Cependant, en temps de résolution total, notre approche n'a pas vraiment l'avantage, puisqu'elle est en dessous de l'approche classique. Elle n'a l'avantage en temps que pour la catégorie "intensions/unsat", dont le temps résolution des instances est le plus faible.

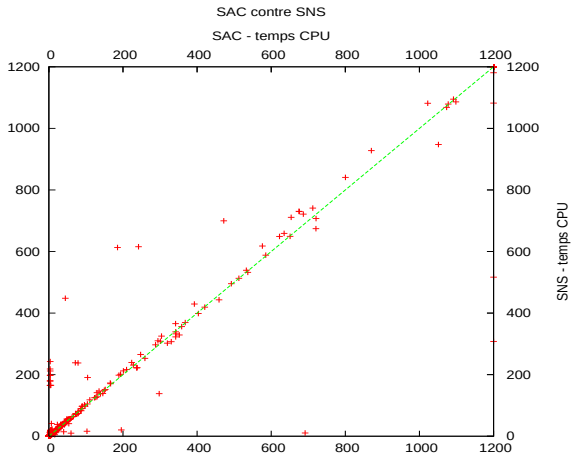


FIG. 1 – Comparaison des temps CPU : SAC contre SNS

Le nuage de points Figure 1 permet d'avoir une vue globale des temps de résolution des instances. Sur celui-ci chacun des points représente un problème, dont les coordonnées sont désignées par les temps de résolution des instances. Par exemple, la droite $x = 1200$ montre les 4 instances résolues par SNS et non par SAC. Il est facile de déduire de cette figure que SNS est plus mauvais en temps, notamment en observant une nuée de point proche de $(0, 200)$. Malgré ces quelques instances, la majorité des points se situent autour de la droite $y = x$, ce qui montre des temps très proches pour les deux approches.

Même si l'approche s'avère mauvaise pour la recherche de performance, certaines classes d'instances ressortent de ces résultats, notamment celles formant le groupement de point énoncé précédemment. Une analyse plus détaillée permet d'extraire deux classes de problème : JobShop et Taillard OpenShop.

Tout d'abord, les problèmes jobShop utilisés, étudiés par Sadeh-Fox, sont des problèmes *sat* de 50 variables, de domaines de taille comprise entre 100 à 250, et de 265 contraintes binaires. La plupart de ces instances ont été résolues facilement par l'approche SAC, sans pour autant bénéficiée de filtrage. Sur 46 instances résolues par les deux approches, 29 sont triviales, c'est-à-dire que le nombre d'affectations équivaut au nombre de variables.

Solver	#	Trival	Temps	Filtr.	Prepro.	Nœuds
SAC	45	29	175	0	85	208181
SNS	45	35	2369	19293	2208	8801

TAB. 2 – Synthèses des résultats des problèmes jobShop

Le tableau 2 présente une comparaison des deux approches étudiées, dont les critères sont le nombre d'ins-

tances résolus (#), le nombre d'instances trivialement satisfaites (Trivial), le temps de résolution en secondes (Temps), le nombre de valeurs filtrées (Filtr.), le temps de prétraitement en secondes (Prepro.) et le nombre de nœuds de l'arbre de recherche (Nœuds). Ceci montre l'efficacité du prétraitement effectué par SNS pour le critère de la suppression de valeurs. Quelques instances sont trivialement satisfaites et le nombre de nœuds liés à la recherche est fortement réduit. Cependant, le temps de prétraitement occupe la majeure partie du temps de résolution (93%), et c'est sur ce critère que notre approche démontre des lacunes. SNS permet de résoudre en $10s$ (dont $8s$ de prétraitement) une $46e$ instance que SAC résoud en $691s$ (dont $2s$ de prétraitement). Notre approche filtre 517 valeurs et réduisant ainsi le nombre de nœuds parcourus à 192, alors que l'approche classique ne permet aucun filtrage, et a parcouru plus de 4 millions de nœuds. Intégré ce résultat à notre synthèse n'aurait donc pas été pertinent.

Pour terminer l'analyse de l'approche statique, étudions la deuxième classe d'instances intéressante : les instances Taillard OpenShop. Ces problèmes d'optimisations sont des instances dont le nombre de variables est n^2, de taille maximale avoisinant les 300 valeurs et le nombre de contraintes est de $4n^2$ pour $n \in \{4, 5, 7, 10, 20\}$ dans notre cas. 43 instances ont été résolues par les deux approches, et celles-ci n'effectuent aucun filtrage pour 18 d'entre elles. Notons que le temps de prétraitement pour ces 18 instances est d'environ $305s$ pour SNS et $303s$ pour SAC, ce qui est intéressant puisque la vérification des états n'a pas été trop coûteuse en temps.

Solver	#	Trivial	Temps	Filtr.	Prepro	Nœuds
SAC	25	3	1161	10850	590	870351
SNS	25	4	1950	67105	1775	151663

TAB. 3 – Synthèses des résultats des problèmes Taillard-OS

Le tableau 3 se composent des mêmes critères que pour les jobShop. Nous pouvons y effectuer les mêmes remarques que précédemment. Le manque d'efficacité en temps vient du prétraitement qui fait un travail de filtrage très important (90% du temps de résolution), pour évaluer un nombre de nœuds bien plus faible.

Chacune des catégories fait ressortir une classe de problèmes particulière, plus ou moins facile à résoudre. Pour autant, il est difficile de déterminer une classe d'instances complète pour laquelle l'approche SNS est meilleure en temps. Notre prétraitement s'avère très efficace pour la détection jointe de valeurs substituables et non-sac, mais ceci a un coût. Bien que le prétraitement soit peu efficace en temps lorsque ce nombre de valeurs est important, il facilite largement la recherche en réduisant fortement le nombre de nœuds de l'arbre, ce qui lui permet de résoudre plus d'instances que son homologue de base.

4.2 Intégration dynamique

Pour notre approche dynamique, les critères de comparaisons se concentrent sur le nombre d'instances résolues, le temps de résolution en seconde et le nombre de filtrages effectués durant la recherche.

Extension SAT

Solveur	#	Temps	# inst.	Filtr.
dual	296	20845	-	-
ns+dual	294	17972	28	129
−dual	294	18600	-	-

Extension UNSAT

Solveur	#	Temps	# inst.	Filtr.
dual	196	13771	-	-
SNS	196	13716	36	66302

Intension SAT

Solveur	#	Temps	# inst.	Filtr.
dual	253	9391	-	-
ns+dual	254	9465	58	22439
−dual	251	9466	-	-
−ns+dual	251	8706	55	18441

Intension UNSAT

Solveur	#	Temps	# inst.	Filtr.
dual	202	8268	-	-
ns+dual	203	8435	29	152014
−ns+dual	202	7789	28	104917

TAB. 4 – Résultats globaux : dual contre ns+dual

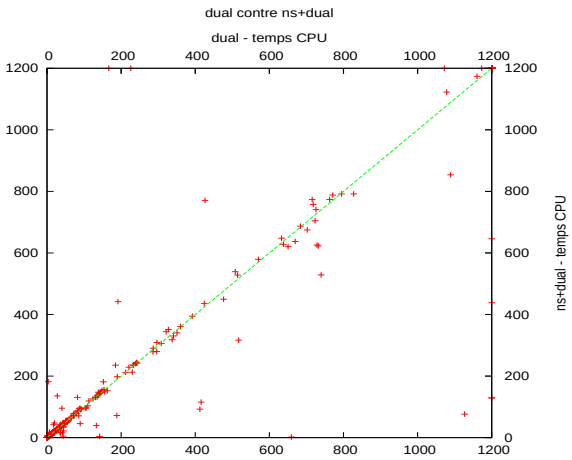


FIG. 2 – Comparaison des temps CPU : dual contre ns+dual

Le tableau 4 montre que le nombre d'instances résolues est le même pour l'une et l'autre des approches. Le cumule des temps ne fournit pas non plus un critère intéressant pour l'approche dynamique, puisque l'on ne note qu'un très léger gain. Néanmoins, pour les instances communes résolues, l'approche ns+dual est toujours meilleure, ce que l'on remarque aussi sur

le nuage de points 2. Un résultat décevant toutefois : 10 à 20% des instances bénéficient de la détection de valeurs substituables.

Malgré le peu d'instances touchées par notre traitement dynamique, il est possible de dégager quelques classes de problèmes, notamment les deux classes étudiées pour l'approche statique. Les jobShop, tout d'abord, sur 46 problèmes résolus par les deux approches, 29 sont trivialement *sat*. Sur les 17 instances restantes, 10 sont touchées par la détection de la substituabilité, dont l'instance que nous avions écarté de notre synthèse. L'approche dual met près de $660s$ pour répondre à la satisfaisabilité, alors que ns+dual, 1 petite seconde, grâce à la détection de 3 valeurs substituables. Le gain obtenu pour les autres instances restent négligeable puisque celles-ci sont faciles ($15s$ pour ns+dual et $17s$ pour dual).

La deuxième classe correspond aux instances Taillard. Sur 45 instances, 25 sont touchées par la détection de la substituabilité de ns+dual dont 2 instances sont résolues uniquement par ns+dual en filtrant 397 valeurs. dual met $1724s$ tandis que ns+dual met $1443s$ pour la résolution des 23 instances restantes, ce qui est expliqué par le filtrage de 2611 valeurs, permettant d'évaluer 460024 nœuds au total, contre 1046687 pour dual. Ceci montre encore une fois l'efficacité de la détection de la substituabilité pour les instances Taillard, et permet de faire le lien, comme pour l'approche statique entre valeurs filtrées et réduction de l'arbre de recherche.

Contrairement à l'approche statique, l'approche dynamique montre de meilleurs résultats en temps qu'un solveur dual classique, même si le nombre d'instances résolues reste le même. Aussi, la non-résolution de quelques instances par notre approche comparé à l'approche classique est due, dans le cas général, à la grande taille de l'instance ($fapp$ avec 2500 variables de taille maximum 360) ou le temps de résolution proche de $1200s$ pour l'approche classique, pour lequel le procédé de détection automatique pénalise notre approche.

5 Conclusion et perspectives

Dans ce papier, une généralisation de la substituabilité au voisinage, une forme de symétrie affaiblie, a été proposée. Cette généralisation originale est obtenue par substitution de la notion de supports de manière syntaxique par une mesure sémantique de la propagation des contraintes définie par l'état des variables dans le voisinage d'une variable affectée. Pour ceci, un prétraitement intégré à l'algorithme SAC a été proposé en tant qu'approche statique. De plus, puisque l'affectation et le maintien de la cohérence d'arc sont des opérations de base des algorithmes de type MAC, une adaptation dynamique a également été proposée. Les résultats expérimentaux sur les problèmes binaires de nos deux approches sont intéressantes par plusieurs aspects. Même si l'approche statique n'est pas rentable en temps, celle-ci se montre d'une grande efficacité de filtrage, ce qui permet la résolution d'instances que l'approche classique ne résoud pas. L'approche dynamique montre des résultats en temps intéressants, même si le nombre de problèmes sur lequel les valeurs substituables ont été détectées est faible. Enfin, certaines classes de problèmes pour lesquelles la détection des valeurs substituables s'avère efficace et pertinente peuvent être extraites et mettent évidence l'impact, positif et négatif, de la détection de la substituabilité au voisinage.

Les perspectives à la suite de ces travaux sont nombreuses. Outre la généralisation et l'amélioration de nos approches en pratique, l'utilisation d'autres formes de propagation des contraintes est envisagée. L'étude de l'impact d'un élagage sur la pondération d'heuristiques est envisagé pour tenter d'avoir une complémentarité. Il nous semble aussi intéressant d'exploiter notre approche pour établir des relations de dépendances entre les valeurs des variables.

Références

[1] Belaid Benhamou and Lakhdar Sais. Tractability through symmetries in propositional calculus. *Journal Automated Reasoning*, 12(1) :89–102, 1994.

[2] James Crawford, Matthew L. Ginsberg, Eugene Luck, and Amitabha Roy. Symmetry-breaking predicates for search problems. In *Principles of Knowledge Representation and Reasoning (KR'96)*, pages 148–159. 1996.

[3] Eugene C. Freuder. Eliminating interchangeable values in constraint satisfaction problems. In *AAAI*, pages 227–233, 1991.

[4] Ian P. Gent and Barbara Smith. Symmetry breaking during search in constraint programming. In *Proceedings ECAI'2000*, pages 599–603, 2000.

[5] Christophe Lecoutre and Sébastien Tabary. Abscon 109 : a generic csp solver. In *2nd International CSP Solver Competition, held with CP'2006*, pages 55–63, 2007.

[6] J.-F. Puget. On the satisfiability of symmetrical constrained satisfaction problems. In *Methodologies for Intelligent Systems : Proc. of the 7th International Symposium ISMIS-93*, pages 350–361. 1993.

Avec le soutien de