Septièmes Journées Francophones de Programmation par Contraintes

afpc jfpc 2011

Lyon, 8-10 juin 2011

Lyon 1

INSA LYON

LIRIS

Actes des Septièmes Journées Francophones de Programmation par Contraintes

JFPC'2011

8-10 Juin 2011, Lyon, France

Organisation

LIRIS -Laboratoire d'InfoRmatique en Image et Systèmes d'information

Présidente des journées

Christine Solnon, LIRIS, Université de Lyon

Président du comité de programme

Arnaud Lallouet, GREYC, Université de Caen Basse-Normandie

Comité de Programme

Gilles Audemard ..... CRIL, Lens
Christian Bessiere ..... LIRMM, Montpellier
Hadrien Cambazard ..... 4C, UCC, Cork, Irlande
Thi-Bich-Hanh ..... LIFO, Orléans
Simon De Givry ..... INRA, Toulouse
Yves Deville ..... UCL, Louvain-la-Neuve, Belgique
François Fages ..... INRIA, Rocquencourt
Carmen Gervet ..... GUC, Le Caire, Égypte
Laurent Granvilliers ..... LINA, Nantes
Youssef Hamadi ..... MSR Inc, Cambridge, UK
Michaël Krajecki ..... CReSTIC, Reims
Christophe Lecoutre ..... CRIL, Lens
Olivier Lhomme ..... IBM, Sophia Antipolis
Xavier Lorca ..... LINA, Nantes
Samir Loudni ..... GREYC, Caen
Florent Madelaine ..... LIMOS, Clermont-Ferrand
Laurent Michel ..... University of Connecticut, Storrs, USA
Eric Monfroy ..... UTFSM, Valparaiso,
Chili Wady Naanaa ..... Université de Monastir, Tunisie
Samba Ndojh Ndiaye ..... LIRIS, Lyon
Richard Ostrowski..... .LSIS, Marseille
Laurent Perron ..... Google Inc, Paris
Cédric Piette ..... CRIL, Lens
Cédric Pralet ..... ONERA, Toulouse
Nicolas Prcovic ..... LSIS, Marseille
Jean-Charles Régin ..... Université de Nice Sophia Antipolis, I3S/ CNRS, Nice
Michel Rueher ..... Université de Nice Sophia Antipolis, I3S/ CNRS, Nice
Lakhdar Sais ..... CRIL, Lens
Frédéric Saubion ..... LERIA, Angers
Pierre Schaus ..... n-Side, Louvain-la-Neuve, Belgique
Pierre Siegel ..... SNC, Toulon
Christine Solnon ..... LIRIS, Lyon
Igor Stefan ..... LERIA, Angers
Charlotte Truchet ..... LINA, Nantes
Élise Vareilles ..... EMN, Albi

Relecteurs additionnels

Slim Abdennadher, Loïc Blet, Jean-Charles Boisson, Rémi Coletta, Alvaro Fialho, David Fournier, Alexandre Goldsztejn, Said Jabbour, Christophe Jaillet, Frédéric Koriche, Jean Marie Lagniez, Matthieu Lopez, Thierry Martinez, Florence Massen, Jordan Ninin, Alexandre Papadopoulos, Marie Pelleau, Quang Dung Pham, Arnaud Renard.

Comité d'organisation

Loïc Blet
Jean-François Boulicaut
Camille Combier
Stéphane Gosselin
Samba Ndojh Ndiaye
Sylvie Oudot
Céline Roardet
Christine Solnon
Laure Tougne

Préface

La Programmation par Contraintes (ou PPC) permet de modéliser et de résoudre aisément des problèmes combinatoires de façon déclarative. Depuis 1997, l'objectif est identique : "L'utilisateur décrit son problème, l'ordinateur le résout" (Gene Freuder, In pursuit of the Holy Grail, Constraints 2(1), 1997). Décrire ou modéliser son problème consiste à trouver les variables et les contraintes liants ces variables. On recherche alors une solution satisfaisant toutes les contraintes, ou dans le cas d'un problème d'optimisation, la meilleure solution satisfaisant les contraintes. Pour cela, le problème est transmis à un solveur qui va effectuer cette recherche.

Les origines de la PPC sont nombreuses et reflètent sa diversité actuelle : la satisfaction de contraintes en Intelligence Artificielle, les aspects langage issus de la Programmation en Logique par Contraintes, la Recherche Opérationnelle forment le cœur des influences.

Les Journées Francophones de Programmation par Contraintes (JFPC) sont avant tout placées sous le signe de la convivialité. Elles permettent à la communauté PPC francophone de se rencontrer et d'échanger. Les actes de cette conférence contiennent les dernières avancées de ce domaine présentées en langue française. Chercheurs, enseignants-chercheurs, étudiants et industriels se retrouvent pour communiquer et visualiser un vaste panorama de l'actualité du domaine, témoignant de la représentativité de la communauté francophone à l'échelon international. Les articles présentés sont soit originaux, soit ont été publiés l'an passé dans les grandes conférences internationales du domaine (IJCAI, AAAI, CP, ECAI, CPAIOR, SAT...).

Les JFPC sont issues de la fusion des conférences JFPLC (Journées Francophones de Programmation en Logique et par Contraintes) nées en 1992 et des JNPC (Journées Nationales sur la résolution pratique des problèmes NP-complets) nées en 1994. Septième édition de la série, les JFPC 2011 à Lyon succèdent à celles de Caen (2010), Orléans (2009), Nantes (2008), Rocquencourt (2007), Nîmes (2006) et Lens (2005).

Les JFPC ont eu le plaisir de recevoir quarante-et-une soumissions, provenant de la communauté francophone en France, Belgique, Maroc, Espagne, Tunisie, Chili, Algérie et Canada. Après une relecture soigneuse effectuée par trois experts, le comité de programme en a sélectionné trente-trois pour une présentation orale. Les thèmes abordés concernent le filtrage et la propagation, la recherche locale, la satisfaction booléenne, le parallélisme, les ASP, les environnements de programmation, sans oublier les applications industrielles.

Les JFPC 2011 sont organisées conjointement avec les JIAF, Journées d'Intelligence Artificielle Fondamentale et ont comme but de faire se rencontrer les deux communautés. Cette rencontre prend la forme d'une session et deux conférences invitées communes, réunissant des communications pouvant intéresser tous les participants. Les conférenciers invités communs sont Patrice Perny (LIP6, UPMC, Paris) sur la théorie de la décision algorithmique et l'optimisation fondée sur les préférences, et Stefan Szeider (TU Wien, Autriche) sur la complexité paramétrisée et les contraintes. De plus, une conférence invitée plus typée JFPC est donnée par Jean-Charles Régin sur le thème "Comment empêcher les arbres de monter au ciel ?".

Je remercie les auteurs des articles soumis pour leur participation à cet évènement annuel témoignant de la vitalité de notre domaine, les membres du comité de programme et les relecteurs additionnels pour leur travail sérieux et constructifs, à Christine Solnon et aux organisateurs qui ont tout mis en œuvre pour que la conférence soit un succès et aux sponsors sans qui l'évènement n'aurait pu avoir lieu: l'IXXI (l'Institut Rhône-alpin des systèmes complexes), le LIRIS, le GDR I3, INSA de Lyon, l'Université Lyon 1, l'INRIA, IBM et Cosytec.

Arnaud Lallouet, président du comité de programme des JFPC 2011

Table des matières

Conférences invitées

Articles

Index des auteurs

Théorie de la décision algorithmique et optimisation fondée sur les préférences

Patrice Perny

LIP6, UPMC, Case 169, 4 place Jussieu, 75005 Paris, France
patrice.perny@lip6.fr

Abstract

Les développements de la théorie de la décision dans les dernières décennies ont fourni une variété de modèles de préférence sophistiqués pour la prise de décision dans des environnements complexes (décision dans l'incertain et le risque, décision multicritère, décision collective). Dans ce domaine, beaucoup d'efforts est consacré à l'analyse axiomatique des préférences et à la justification théorique des modèles mathématiques, à des fins descriptives ou normatives. Dans la plupart des cas, moins d'attention est consacrée à la détermination des solutions préférées, qui se déduisent souvent facilement du modèle. Toutefois, lorsque l'ensemble des solutions réalisables a une structure combinatoire ou est défini implicitement (par exemple par un ensemble de contraintes), la spécification du modèle de préférence n'est pas suffisante. La détermination des solutions préférées pose des problèmes computationnels et soulève des questions algorithmiques nouvelles en raison de la structure particulière des modèles de préférence. Par exemple, certaines approches constructives simples utilisées en optimisation combinatoire (la programmation dynamique, l'algorithme glouton) ne s'étendent pas automatiquement pour travailler avec des modèles de préférences non-standard. Ceci suggère de revisiter les problèmes d'optimisation classiques abordés dans les manuels de recherche opérationnelle, à la lumière de la décision théorie de la décision, et de développer de nouveaux algorithmes pour l'optimisation fondée sur les préférences. L'objectif de cette présentation est de fournir un aperçu des problèmes étudiés actuellement dans cette direction en Théorie de la décision algorithmique. Nous utiliserons des exemples choisis dans divers contextes tels que la recherche de compromis en optimisation multiobjectif, l'optimisation équitable pour la prise de décision multi-agents, l'optimisation robuste et la décision dynamique dans l'incertain.

Comment empêcher les arbres de monter au ciel

Jean-Charles Régin

Université de Nice-Sophia Antipolis
Parc Valrose
06108 Nice Cedex 2 - France
Jean-Charles.REGIN@unice.fr

Abstract

Dans cet exposé nous essaierons de prendre un peu de recul par rapport à la programmation par contraintes, d'un point de vue à la fois théorique et pratique. Nous étudierons tout d'abord son positionnement par rapport à la question majeure de la complexité (P=NP ?). Nous rappellerons que si NP n'est pas égal à P alors nous ne pourrons jamais résoudre certains problèmes et que nous ne pouvons pas faire mieux que retarder l'explosion exponentielle du temps de réponse. En partant de ce constat, nous montrerons que la recherche en CP n'a donc de sens que si elle a pour but d'améliorer la résolution de problèmes réels. Nous considérerons alors plusieurs exemples de recherche théorique qui se sont avérés pertinents en pratique. Ensuite, nous nous intéresserons à ce que l'on nomme habituellement la recherche appliquée. Nous expliquerons pourquoi il ne faut jamais perdre de vue la résolution de problèmes réels et nous évoquerons certains pièges à éviter (la recherche doit être à la fois généralisable et applicable). Notre exposé sera illustré de nombreux exemples.

Parameterized Complexity and Constraints

Stefan Szeider

TU Wien
Favoritenstraße 9-11
A-1040 Vienna, Austria
stefan@szeider.net

Abstract

In this talk I will advocate the framework of *parameterized complexity* for the theoretical analysis of problems that arise in constraint programming and satisfiability. Parameterized complexity considers in addition to the input size of a problem also a secondary measurement, the parameter. The parameter can represent a qualitative aspect of the input, e.g., how well it is structured. This two-dimensional setting allows a more fine-grained complexity analysis with the potential of being more realistic than the one-dimensional setting of classical complexity theory.

The central notion of parameterized complexity is *fixed-parameter tractability* (FPT) which refers to solvability in time $f(k)n^c$, where n denotes the input size, f is some function (usually exponential) of the parameter k, and c is a constant. The subject splits into two complementary questions, each has its own theoretical toolkit and methods : how to design and improve fixed-parameter algorithms, and how to gather evidence that a problem is not fixed-parameter tractable for a certain parameter.

In this talk I will discuss the basic notions of parameterized complexity and review some recent results on constraint satisfaction, global constraints, and satisfiability. I will mainly focus on three groups of parameters : parameters that represent the distance from a tractable subproblem, parameters that represent decomposability, and parameters that represent locality. Furthermore, I will discuss the concept of *kernelization*, a central concept of parameterized complexity, which can be seen as a form of polynomial-time preprocessing with performance guarantee.

Vers une gestion fine et dynamique de la base de clauses apprises

Gilles Audemard, Jean-Marie Lagniez, Bertrand Mazure, Lakhdar Saïs *

Université Lille-Nord de France
CRIL - CNRS UMR 8188
Artois, F-62307 Lens

{audemard,lagniez,mazure,sais}@cril.fr

Résumé

Dans ce papier, nous proposons une nouvelle heuristique dynamique pour la gestion de la base de clauses apprises dans le cadre des solveurs SAT modernes. Cette approche est basée sur le principe d'activation et de désactivation de clauses. Etant donnée une étape de la recherche, nous utilisons une fonction pour activer certaines clauses et en geler d'autres. De cette manière certaines clauses inutiles à un moment de la recherche pourront de nouveau utilisées dans le futur. Notre stratégie tente d'exploiter certaines informations du passé pour prédire si une clause sera ou non utilisée dans le futur. Elle diffère des autres stratégies par le fait que les clauses ne sont pas supprimées définitivement. Les expérimentations faites sur les instances de la dernière compétition SAT montrent l'efficacité de l'approche proposée.

Abstract

In this paper, we propose a new dynamic management policy of the learnt clauses database in modern SAT solvers. It is based on a dynamic freezing and activation principle of the learnt clauses. At a given search state, using a relevant selection function, it activates the most promising learnt clauses while freezing irrelevant ones. In this way, clauses learned in previous steps can be discarded at the current step and might be activated again in future steps of the search process. Our strategy tries to exploit pieces of information gathered from the past to deduce the relevance of a given clause for the remaining search steps. This policy contrasts with all the well-known deletion strategies, where a given learned clause is definitely eliminated. Experiments on SAT instances taken from the last competitions demonstrate the efficiency of our technique.

*Ce travail est supporté par l'ANR « UNLOC », ANR 08-BLAN-0289-01.

1 Introduction

Le problème SAT est le problème de décision qui consiste à vérifier si une formule booléenne, souvent représentée sous Forme Normale Conjonctive (CNF), admet ou non une affectation de ses variables qui associe « vrai » à la formule. Ce problème est très important en théorie de la complexité (problème NP-Complet de référence) et a beaucoup d'application en intelligence artificielle (planification, raisonnement non monotone, bioinformatique, vérification formelle ...).

Au cours de ces deux dernières décennies le problème SAT, avec l'arrivée d'une nouvelle génération de solveurs, a énormément gagné en intérêt. Ces solveurs, appelés CDCL [11, 5] (Conflict Driven, Clause Learning) ont pour la première fois été capables de résoudre des instances conséquentes codant des problèmes de la vie réelle. Leur architecture est basée sur les notions suivantes : l'heuristique de choix de variables VSIDS (Variable State Independant, Decading Sum) [11], l'utilisation de redémarrage [6, 7] et d'une structure de données efficace (*ex.* Watched literals) ainsi que l'apprentissage de clauses [9, 10, 15]. Cette dernière est une composante très importante de ce type de solveur. En effet d'un point de vue théorique, K. Pipatsrisawat et A. Darwiche [13] ont récemment prouvé que l'apprentissage de clauses tel qu'il est fait dans les démonstrateurs SAT modernes permet de simuler la résolution générale. Ce résultat montre que l'apprentissage rend les démonstrateurs de type DPLL [4] aussi puissants que la résolution générale. Mais en pratique, cet ajout de clauses peut poser pro-

blème. En effet, dans le pire des cas, la taille de la base de connaissance peut croître de manière exponentielle. Pour maintenir une base des clauses apprises de taille polynomiale - permettant ainsi un coût raisonnable pour la propagation unitaire - un ensemble de stratégies a été proposé pour supprimer les clauses supposées inutiles pour la suite de la recherche. Parmi ces stratégies, deux ont montré des résultats pratiques intéressants. La première, la plus populaire, est basée sur la notion de *first fail*. Cette stratégie considère qu'une clause apparaissant peu dans l'analyse des conflits récents est inutile et qu'elle doit être supprimée. La seconde est basée sur une mesure nommée LBD (*Litetal Block Distance*) [1]. Cette mesure statique correspond au nombre de niveaux différents intervenant dans la génération de la clause apprise. Les auteurs considèrent alors que les clauses ayant un LBD élevés sont inutiles pour la suite de la recherche et qu'elles peuvent donc être supprimées. Le principal inconvénient de ces stratégies est qu'il est impossible d'éviter la suppression de clauses potentiellement utiles pour la suite de la recherche. Leur élimination de manière définitive rend possible la dérivation répétitive de la même clause.

Dans ce papier, nous proposons une nouvelle heuristique dynamique pour la gestion de la base des clauses apprises dans le cadre des solveurs SAT modernes. Cette approche s'appuie sur le principe d'activation et de désactivation de clauses. Étant donnée une étape de la recherche, nous utilisons une fonction pour activer certaines clauses et en désactiver d'autres. Cette fonction, basée sur le *progress saving* [12], tente d'exploiter certaines informations du passé pour prédire si une clause sera ou non utilisée dans le futur. L'idée de notre approche consiste alors à geler une clause quand elle est jugée inutile pour la recherche en cours, et à la réactiver lorsqu'elle peut jouer un rôle dans l'établissement de la preuve. Cette stratégie diffère des stratégies habituelles puisque les clauses ne sont pas définitivement supprimées.

La suite de ce papier est organisée de la manière suivante : après quelques définitions préliminaires (section 2), nous introduirons une nouvelle mesure, basé sur le *progress saving*, permettant d'évaluer l'utilité d'une clause (section 3). Ensuite, nous présentons une approche dynamique de nettoyage de la base des clauses apprises basée sur la notion d'activation et de désactivation de clauses (section 4). Avant de conclure et de donner quelques perspectives, nous comparons expérimentalement notre politique de nettoyage des clauses apprises avec les politiques de nettoyage faisant parti de l'état de l'art (section 5).

2 Définitions, notations et notions préliminaires

Dans cette section, après quelques définitions et notations, nous introduisons le schéma général d'un solveur CDCL.

Une formule CNF Σ est une conjonction (interprétée comme un ensemble) de *clauses*, où une clause est une disjonction (interprétée comme un ensemble) de *littéraux*. Un littéral est une variable positive x ou négative $\neg x$. Les deux littéraux x et $\neg x$ sont dit *complémentaires*. Une clause unitaire est une clause possédant un unique littéral (appelé *littéral unitaire*). Une clause vide, notée $\bot$, est interprétée comme fausse, tandis que la *formule* CNF *vide*, notée $\top$, est interprétée comme vraie. Une ensemble de littéraux est *complet* s'il contient un littéral pour chaque variable apparaissant dans Σ et est *fondamental* s'il ne contient pas un littéral et son complémentaire. Une *interprétation* $\mathcal{I}$ d'une formule booléenne Σ associe une valeur $\mathcal{I}(x)$ à certaines variables x de la formule Σ. Une interprétation peut être représentée par un ensemble de littéraux complet et fondamental. Un *modèle* d'une formule Σ est une interprétation $\mathcal{I}$ qui satisfait la formule, *i.e.* toutes les clauses de la formule. Finalement, SAT est le problème de décision qui consiste à vérifier si une formule Σ sous Forme Normale Conjonctive (CNF) possède ou non un modèle.

Algorithm 1: CDCL solver

```
   Input: a CNF formula Σ
   Output: SAT or UNSAT
1  Δ = ∅;   /* learnt clauses database */
2  while (true) do
3    if ( !propagate()) then
4      if ((c = analyzeConflict()) == ∅) then
       return UNSAT ;
5      Δ = Δ ∪ {c};
6      if (timeToRestart()) then backtrack to level 0;
7      else
8        backtrack to the assertion level of c;
9    else
10     if ((l = decide()) == null) then return SAT ;
11     assert l in a new decision level;
12     if (timeToReduce()) then clean(Δ);
```

Nous présentons brièvement le schéma général d'un solveur SAT de type CDCL [11, 5]. Ces solveurs incorporent la propagation unitaire, une heuristique de choix de variable basée sur l'activité, une heuristique de choix de polarité, l'apprentissage de clauses basé sur l'analyse du graphe d'implication, une stratégie de redémarrage et une politique de réduction de la base de clauses apprises.

L'algorithme 1 donne le schéma général d'un solveur de type CDCL. Typiquement, un tel solveur peut être assimilé à une séquence de décision suivie de propagations des littéraux unitaires, jusqu'à l'obtention d'un conflit. Chaque littéral choisi comme littéral de décision (ligne 10) est affecté suivant une certaine polarité à un niveau de déci-

sion donné (ligne 11). Si tous les littéraux sont affectés ($decide() == null$), alors $\mathcal{I}$ est un modèle de Σ (lignes 10). À chaque fois qu'un conflit est atteint par propagation (ligne 3), une clause (*nogood*) c est calculée en utilisant une méthode d'analyse de conflits donnée, le plus souvent selon le schéma « First UIP » (*« Unique Implication Point »* [15]), et un niveau de *backjump* est calculé en fonction de la clause c. À ce moment, il est possible de prouver l'inconsistance (c est la clause vide) de la formule (ligne 4). Si ce n'est pas le cas, un saut arrière est effectué et le niveau de décision devient égal au niveau de *backjump* (ligne 8). Ensuite, certains solveurs CDCL forcent le redémarrage (diverses stratégies sont possibles [7]) et dans ce cas, on remonte en haut de l'arbre de recherche (ligne 6). Finalement, la base de clauses apprises peut être réduite lorsque celle-ci est considérée comme trop volumineuse (ligne 12).

Cette dernière composante, bien que souvent omise de la description des démonstrateurs CDCL, est pourtant cruciale pour leurs performances. En effet, conserver un grand nombre de clauses peut avoir des conséquences néfastes sur l'efficacité de la propagation unitaire, tandis que supprimer trop de clauses peut faire perdre le bénéfice de l'apprentissage. Par conséquent, identifier les bonnes clauses apprises (*i.e.* importantes pour la dérivation de la preuve) est un véritable challenge. La première mesure proposée pour quantifier la qualité d'une clause utilise l'idée sous-jacente à l'heuristique VSIDS. Plus précisément, une clause apprise est considérée comme utile pour la preuve, si elle apparaît fréquemment dans l'analyse des conflits récents. On remarque donc qu'une telle stratégie suppose qu'une clause utile dans le passé le sera dans le futur. Plus récemment, une nouvelle mesure, nommée LBD, a été utilisée pour estimer la qualité d'une clause apprise. L'implémentation de cette mesure dans un solveur CDCL (le solveur GLUCOSE [1]) a permis d'en améliorer sensiblement les performances. Cette nouvelle mesure est basée sur le calcul du nombre de niveaux de décision apparaissant dans une clause apprise lors de sa génération. Les auteurs on démontré expérimentalement que les clauses possédant un faible LBD sont plus souvent utilisées pendant la recherche que les clauses ayant un LBD élevé.

Un autre point crucial des solveurs CDCL concerne la phase d'affectation selon laquelle la prochaine variable choisie par l'heuristique VSIDS sera affectée. Habituellement, une phase par défaut (*e.g.* fausse) est définie et utilisée chaque fois qu'une variable de décision est assignée. Récemment, Pipatsrisawat et Darwiche [12] ont observé expérimentalement que les solveurs implémentant les redémarrages et les retours arrière non chronologique, peuvent être amenés à résoudre de manière répétitive les mêmes sous-problèmes. Pour éviter cela, les auteurs ont proposé une heuristique dynamique consistant à sauvegarder la phase selon laquelle une variable a été propagée, et à utiliser cette information lors des prochaines affectations de cette variable. Cette heuristique de phase, nommée *progress saving*, permet d'éviter la résolution répétitive des mêmes sous-formules satisfiables. Cette mémorisation de la phase peut être représentée par une interprétation complète $\mathcal{P}$. De cette manière, lorsqu'une nouvelle variable v est choisie comme décision, le littéral ℓ qui lui est associé (pour la phase) est choisi tel que $\ell \in \mathcal{P}$.

Dans la section suivante, nous exploitons la notion de *progress saving* ($\mathcal{P}$) pour estimer si une clause peut potentiellement être utile pour dériver une clause apprise (*i.e.* la possibilité qu'elle soit utilisée pour propager un littéral menant au conflit).

3 Une nouvelle mesure pour identifier les bonnes clauses

Comme précisé précédemment les solveurs SAT de type CDCL peuvent formaliser un système de preuve par résolution [13, 2]. Le principal inconvénient des méthodes basées sur la résolution générale est lié à leur complexité spatiale. D'un certain point de vue, les solveurs SAT modernes peuvent être vus comme une méthode de preuve par résolution avec une stratégie de suppression de clauses. Par conséquence, la complétude des solveurs SAT modernes est fortement liée à la politique de suppression de clauses. Par exemple, supposons que la stratégie de nettoyage de la base de clauses apprises soit très agressive, dans ce cas il nous serait impossible de garantir la complétude du solveur. Définir quelle clause sera utile pour la preuve est d'une grande importance pour l'efficacité des solveurs. Répondre à une telle question est cependant calculatoirement difficile et est très proche du problème qui consiste à trouver une preuve de taille minimale. Pour apporter une solution à ce problème nous définissons une mesure simple permettant d'évaluer la pertinence d'une clause et montrons expérimentalement sont efficacité. La mesure proposée dans cette section est basée sur la notion de *progress saving* [12], habituellement utilisée pour déterminer la valeur de vérité dont sera affectée la prochaine variable de décision. Cette mesure, nommée psm (*Progress Saving based quality Measure*), est définie de la manière suivante : étant données une clause c et une interprétation complète $\mathcal{P}$ représentant l'ensemble des polarité associées à chaque variable, nous définissons $psm_{\mathcal{P}}(c) = |\mathcal{P} \cap c|$.

Premièrement, il est important de noter que notre mesure *psm* est dynamique. En effet, puisque l'ensemble des littéraux $\mathcal{P}$ associés à la polarité des variables évolue durant la recherche, le *psm* d'une clause donnée évolue par la même occasion. Par exemple, quand une clause est apprise son *psm* est égal à zéro, et lorsque le retour-arrière est effectué son *psm* devient égal à un. Il est aussi important de noter que lorsqu'une clause est à l'origine de la propagation d'un littéral, son *psm* est aussi égal à un. Ces remarques préliminaires suggèrent que les clauses possédant une petite valeur

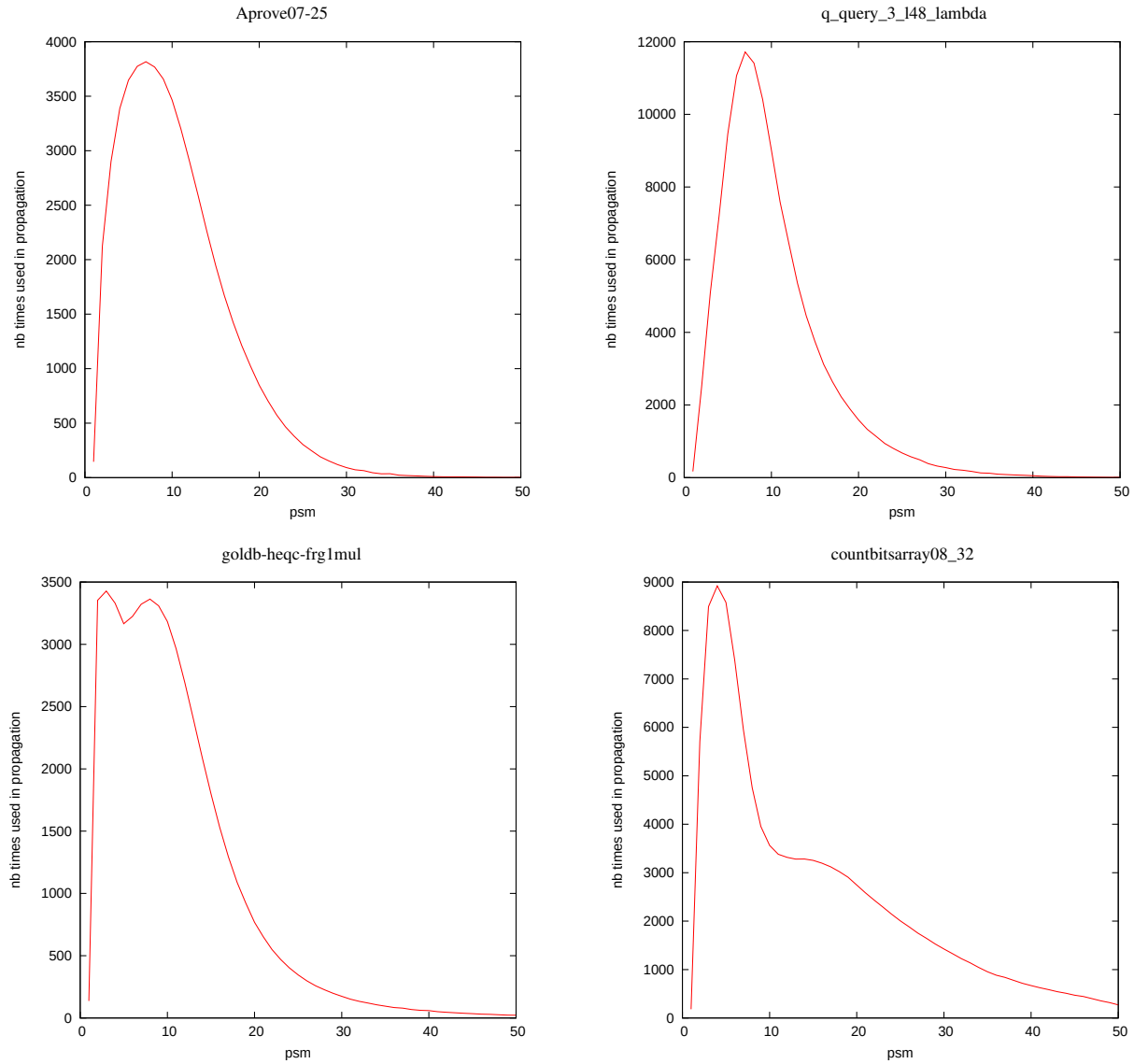


FIGURE 1 – Progress Saving mesure / utilité dans la PU

de *psm* sont plus importantes pour la suite de la recherche. En effet, considérons $\mathcal{I}$ l'interprétation partielle courante, $\mathcal{P}$ l'interprétation complète représentant le *progress saving* et c une clause. Comme $\mathcal{I} \subset \mathcal{P}$, $psm_{\mathcal{P}}(c)$ représente le nombre de littéraux affectés à vrai par $\mathcal{I}$ ou qui seront affectés à vrai par $\mathcal{P} \backslash \mathcal{I}$. Donc, une clause avec un petit *psm* a de fortes chances d'être utilisée pour la propagation unitaire ou d'être falsifiée. Au contraire, une clause avec un grand *psm* a plus de chance d'être satisfaite par plus d'un littéral et donc d'être inutile pour la suite de la recherche.

Pour analyser et valider cette hypothèse, un ensemble d'expérimentations a été mené. La Figure 1 reporte, pour quelques instances représentatives, la moyenne du nombre de fois où une clause avec une certaine valeur de *psm* a été utilisée durant le processus de propagation. Dans cette expérimentation, nous considérons comme intervalle de temps le moment de la recherche où la base de clauses apprises est nettoyée. Cet intervalle de temps est noté t_k avec $k > 0$ (le début de la recherche est à t_0). Soient $\mathcal{P}_{t_k}$ et $\mathcal{P}_{t_{k+1}}$ les interprétations représentant respectivement le *progress saving* aux étapes t_k et t_{k+1}. Considérons la fenêtre de temps entre t_k et t_{k+1} : lorsqu'une clause c provenant de la base de clauses apprises est utilisée dans le processus de propagation unitaire nous calculons $psm = psm_{\mathcal{P}_{t_k}}(c)$ et incrémentons la variable $\alpha(psm)$ (qui correspond au nombre de fois où une clause avec une telle valeur de *psm* est utilisée pour propager un littéral). Le nombre moyen de fois où une clause possédant un *psm* donné (axe des x) est utilisé dans le processus de propagation unitaire (axe des y), correspond alors à $\alpha(psm)$ divisé par le nombre total de fois où la base de clauses apprises a été nettoyée.

D'après la Figure 1, les clauses apprises possédant une petite valeur de *psm* sont plus souvent utilisées dans le processus de propagation que les clauses possédant une grande valeur de *psm*. Si nous regardons plus en détail, nous pouvons voir que les clauses les plus utilisées ont une valeur de *psm* proche de 10. Nous pouvons assurer, puisque l'expérimentation a été menée sur un très large panel d'instances, que dans la majorité des instances considérées la distribution de la valeur de *psm* ressemble à celle des deux courbes hautes de la Figure 1.

Cette première expérimentation illustre le fait que les clauses possédant une faible valeur de *psm* sont importantes dans le processus de propagation. Pour valider de manière expérimentale notre mesure nous avons choisi de l'intégrer dans une politique de réduction de la base de clauses apprises (clean(Δ) - ligne 12 de l'algorithme 1) et de comparer celle-ci avec les approches de réduction de clauses apprises existantes dans la littérature. Cette nouvelle stratégie de réduction de base a été introduite dans le solveur MINISAT 2.2. Comme pour les approches de réduction classique, l'ensemble des clauses apprises est tout d'abord trié par ordre croissant (ici, en fonction de leur *psm*). Lorsque deux clauses possèdent le même *psm*, nous utilisons la stratégie classique basée sur l'activité (VSIDS) pour les départager. Une fois les clauses triées la base de clauses apprises est réduite de moitié. Comme pour les autres stratégies, nous conservons toujours les clauses binaires.

Dans la suite, toutes nos expérimentations ont été menées sur un Quad-core Intel XEON X5550 avec 32Gb de mémoire. 292 instances ont été utilisées, elles sont issues de la compétition SAT 2009, catégorie industrielle. Pour l'ensemble des expérimentations le temps CPU est limité à 900 secondes.

Pour chaque solveur, nous indiquons le nombre d'instances résolues (#Résolues) ainsi que le nombre d'instances résolues satisfiables (#SAT) et insatisfiables (#UNSAT). Nous donnons aussi la moyenne de temps nécessaire pour résoudre une instance (tps moy).

Solveur	#Résolues (#SAT- #UNSAT)	tps moy
MINISAT^d	174 (68 - **106**)	142
MINISAT^d + *psm*	**177** (**73** - 104)	130
MINISAT^d + LBD	173 (71 - 102)	132

TABLE 1 – Résultats avec la séquence de réduction utilisée par défaut dans le solveur MINISAT.

La tableau 1 résume les résultats obtenus pas MINISAT^d [5], MINISAT^d + LBD [1] et MINISAT^d + psm utilisant la séquence de réduction proposée par défaut (notée MINISAT^d) dans MINISAT 2.2. Comme nous pouvons le voir, MINISAT^d+psm est le solveur qui résout le plus d'instances. C'est également le meilleur sur les instances SAT. Cette expérimentation montre l'efficacité de notre mesure dans le cas d'une politique utilisant la séquence de réduction proposée par défaut dans MINISAT 2.2.

Solveur	#Résolues (#SAT- #UNSAT)	tps moy
MINISAT^a	162 (68 - 94)	136
MINISAT^a + *psm*	163 (70 - 93)	140
MINISAT^a + LBD	**168** (**72** - **96**)	128

TABLE 2 – Résultats avec la séquence de réduction utilisée agressive.

Pour être plus juste dans la comparaison des trois approches, nous présentons aussi dans le tableau 2 les résultats obtenus lorsque la séquence de réduction est plus agressive, comme celle utilisée dans [1] (noté MINISAT^a). Dans cette expérimentation la séquence de réduction de la base de clauses apprises Δ est calculée de la manière suivante : $t_0 = 4000$ conflits et $t_k = t_{k-1} + 300$ conflits pour $k > 0$. En utilisant une stratégie de réduction plus agressive, les résultats obtenus par le LBD sont meilleurs que ceux obtenus à l'aide de la stratégie VSIDS et par notre mesure *psm*.

Pour résumer, ces premières expérimentations ont clairement montré la viabilité de notre nouvelle mesure. Cette dernière étant compétitive avec les autres méthodes existant dans la littérature.

4 Geler pour ne pas supprimer : une politique dynamique de réduction de la base de clauses

Dans la section 3, nous avons défini une nouvelle mesure basée sur la notion de *progress saving* pour identifier les clauses apprises importantes. Dans cette section, nous décrivons une approche dynamique de gestion de la base de clauses apprises. Cette approche est basée sur deux notions-clés. Premièrement, le *progress saving* évolue durant la recherche. Par conséquent, une clause peut être considérée comme inutile (avec une grande valeur de *psm*) à un instant de la recherche et devenir intéressante (avec une petite valeur de *psm*) à un autre moment de la recherche. Deuxièmement, déterminer si une clause donnée sera importante pour la preuve est un problème calculatoirement difficile. Toutes les politiques de nettoyage de la base de clauses apprises proposées dans la littérature ne sont pas sûres de ne pas supprimer une clause importante pour la suite de la recherche. Pour ces deux raisons, l'approche proposée introduit un nouveau concept qui consiste à geler certaines clauses. Lorsqu'une clause est considérée comme inutile à un instant de la recherche celle-ci devient gelée et sera réactivée lorsqu'elle sera de nouveau considérée comme importante. Plus précisément, geler (respectivement activer) une clause signifie que la clause est déconnectée (respectivement connectée) à la base des clauses

apprises et ne participe donc pas à la recherche (propagation unitaire...).

Cette politique de nettoyage de la base de clauses apprises basée sur le principe de désactivation et d'activation de clauses ne peut pas être utilisée avec les mesures habituelles. En effet, le LBD d'une clause apprise est statique car défini à la création de la clause, et ne change pas durant la recherche. Tandis que la mesure basée sur l'activité (basée sur VSIDS) est dynamique mais peut seulement mettre à jour les clauses attachées (les clauses doivent être utilisées pour être pondérées).

À présent, décrivons de manière formelle notre politique de nettoyage de la base de clauses apprises. Premièrement, comme le *psm* d'une clause est fortement dynamique, nous introduisons la notion de *déviation* entre deux *progress saving* successifs. Soit V_{t_k} l'ensemble des variables assignées par le solveur entre l'étape t_{k-1} et t_k. La déviation d_{t_k} est définie de la manière suivante : $d_{t_k} = \frac{h(\mathcal{P}_{t_k}, \mathcal{P}_{t_{k-1}})}{|V_{t_k}|}$, où h donne la distance de Hamming entre deux interprétations. Cette notion de déviation est une normalisation de la distance de Hamming pour ne prendre en compte que les variables réellement utilisées par le solveur. Elle permet de calculer l'évolution de l'interprétation liée au *progress saving* entre deux nettoyages de base successifs. Une déviation qui tend vers zéro indique que le solveur continue d'explorer le même espace de recherche, alors qu'une déviation importante indique que le solveur explore une partie différente de l'espace de recherche.

Pour obtenir une vue plus précise du comportement de la déviation, nous introduisons la notion de déviation minimale d^m. À l'instant t_k la déviation minimale est donnée par : $d^m_{t_k} = min\{d_{t_i} | 0 \leq i \leq k\}$.

L'utilisation de cette notion de déviation minimale nous permet d'affiner notre mesure *psm*. En effet, soit c une clause évaluée à l'étape t_k, si $psm_{\mathcal{P}_{t_k}}(c) > d^m_{t_k} \times |c|$ alors la clause c à de fortes chances d'être satisfaite dans le futur, sinon elle sera sûrement utilisée dans le processus de propagation.

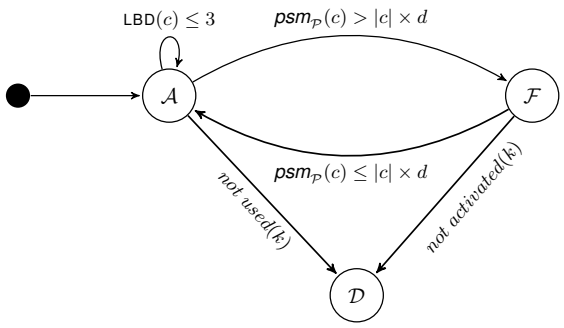


FIGURE 2 – Cycle de vie d'une clause apprise.

La Figure 2 décrit notre approche sous la forme d'un automate d'état. À chaque étape de nettoyage t_k, les clauses peuvent passer d'un état à un autre suivant certaines conditions.

Pour commencer, notons qu'une clause apprise c possède trois états :

1. *État activé* $\mathcal{A}$: c est activée et attachée, elle est utilisée par le solveur ;
2. *État gelé* $\mathcal{F}$: c est gelée *i.e.* c n'est pas attachée ;
3. *État mort* $\mathcal{D}$: c est supprimée.

À présent, décrivons les différentes transitions de l'automate à état.

- Chaque fois qu'une nouvelle clause est apprise celle-ci entre dans l'état $\mathcal{A}$;
- Une clause $c \in \mathcal{A}$ avec une valeur de LBD ($lbd(c) \leq 3$ sur la figure) reste dans l'état $\mathcal{A}$ jusqu'à la fin du processus de recherche ;
- Une clause $c \in \mathcal{A}$ telle que $\frac{psm_{\mathcal{P}_{t_k}}(c)}{|c|} > d^m_{t_k}$ passe à l'état gelé $\mathcal{F}$;
- Une clause $c \in \mathcal{F}$ telle que $\frac{psm_{S_{n_i}}(c)}{|c|} \leq d^m_{n_i}$ passe à l'état activé $\mathcal{A}$;
- Une clause $c \in \mathcal{F}$ qui n'a pas été activée pendant k étapes de nettoyage est supprimée. De manière similaire, une clause $c \in \mathcal{A}$ restant active pendant plus de k étapes sans participer à la recherche est aussi supprimée. Dans les deux cas, cette clause entre dans l'état $\mathcal{D}$. Pour notre étude expérimentale la valeur de k a été fixée à 7.

Un des principaux avantages de notre méthode réside dans le fait qu'il est possible d'augmenter la fréquence de nettoyage de la base de clauses apprises sans l'inconvénient de supprimer une clause importante pour la suite de la recherche. Nous pouvons donc utiliser une politique de nettoyage de base très agressive. Nous avons choisi $t_0 = 500$ et $t_{k+1} + 100$ conflits.

Pour valider notre approche, nous avons étudié expérimentalement le comportement des clauses au sein de l'automate à état. La Figure 3 montre, pour les mêmes instances que celles considérées dans la Figure 1, le nombre de clauses actives, gelées et supprimées ainsi que le nombre de transitions de l'état actif à l'état gelé et vice versa. Ces différentes données sont représentées sur l'axe y des ordonnées tandis que sur l'axe des abscisses x représente le nombre d'opérations de nettoyage de la base de clauses apprises. Pour des raisons de lisibilité, l'ensemble des courbes a été lissé. Pour l'ensemble de ces instances, le nombre de clauses gelées ($Frozen$) et le nombre de clauses actives ($Active$) sont relativement similaires. En ce qui concerne le taux de transfert d'un état à un autre, nous pouvons remarquer que le nombre de clauses passant de l'état actif à l'état gelé ($\mathcal{A} \rightarrow \mathcal{F}$) est plus grand que le nombre de clauses passant de l'état gelé à l'état actif ($\mathcal{F} \rightarrow \mathcal{A}$). Ceci s'explique par le fait que la base des clauses apprises est

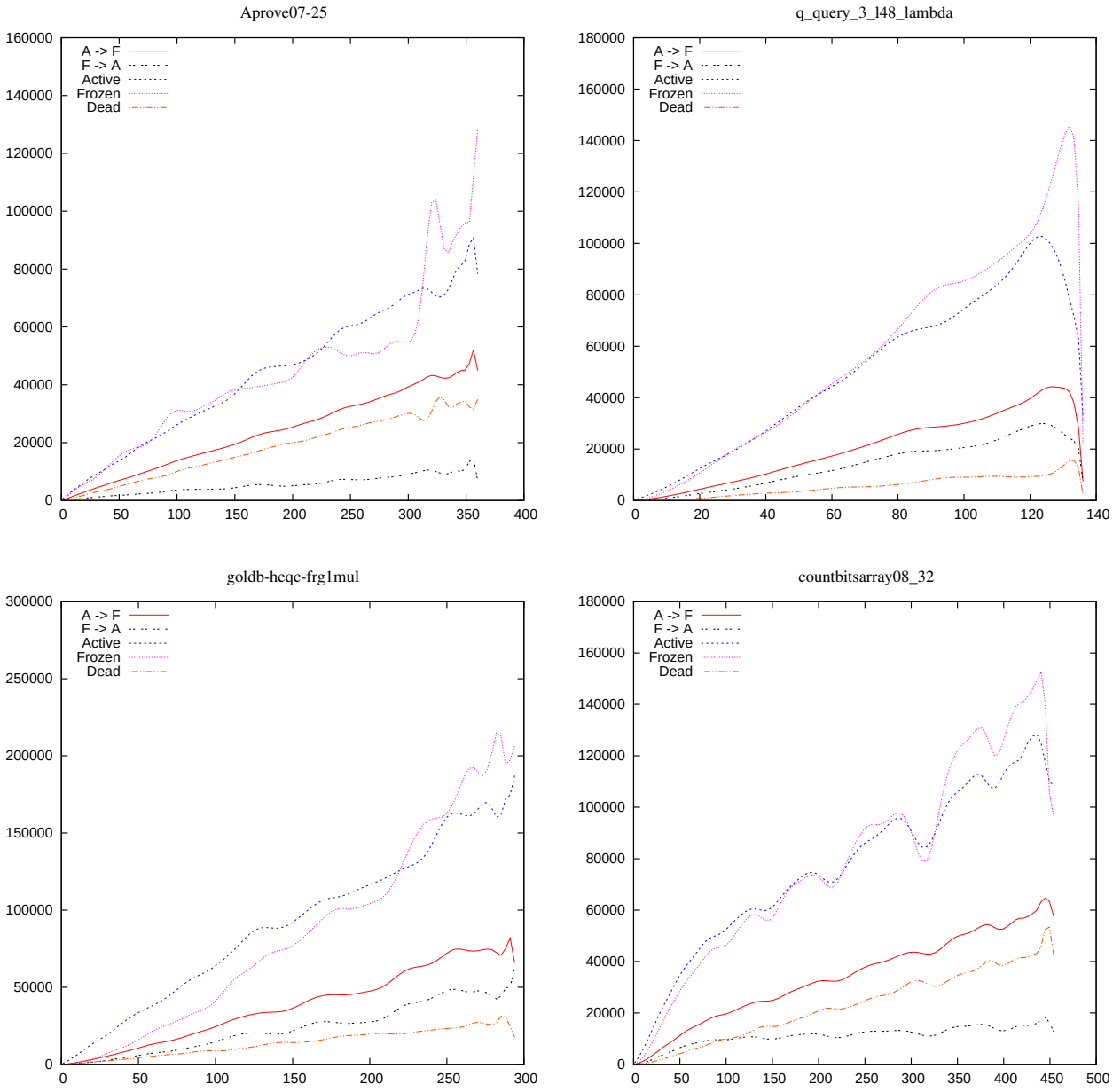


FIGURE 3 – Étude du taux de transfert.

sans cesse alimentée par de nouvelles clauses (ces clauses sont initialement dans l'état actif $\mathcal{A}$). Pour terminer, nous pouvons observer qu'à chaque étape du nettoyage certaines clauses sont définitivement supprimées ($Dead$).

5 Expérimentations

Cette section est divisée en deux parties. Dans la première partie, nous comparons notre approche dynamique de nettoyage de la base de clauses apprises (psm_{dym}) vis-à-vis des approches de la littérature (LBD, VSIDS) décrites dans la section 3. Dans la seconde partie, nous comparons notre approche avec trois autres solveurs de l'état de l'art : (i) le solveur GLUCOSE qui utilise le LBD pour choisir les clauses à conserver ou à supprimer lors de la réduction de la base de clauses apprises, une stratégie de redémarrage dynamique et d'autres composantes décrites dans [1] ; (ii) le solveur LINGELING, dont un ensemble d'améliorations permet d'augmenter la puissance de raisonnement (*ex :* les clauses bloquées [8]) ; et finalement, (iii) le solveur CRYPTOMINISAT, qui implémente nombre des améliorations se trouvant dans la littérature (*ex :* vivification, raisonnement sur les clauses xor ...). La description de ces trois solveurs est disponible sur le site de la SATRACE 2010 (`http://baldur.iti.uka.de/sat-race-2010`).

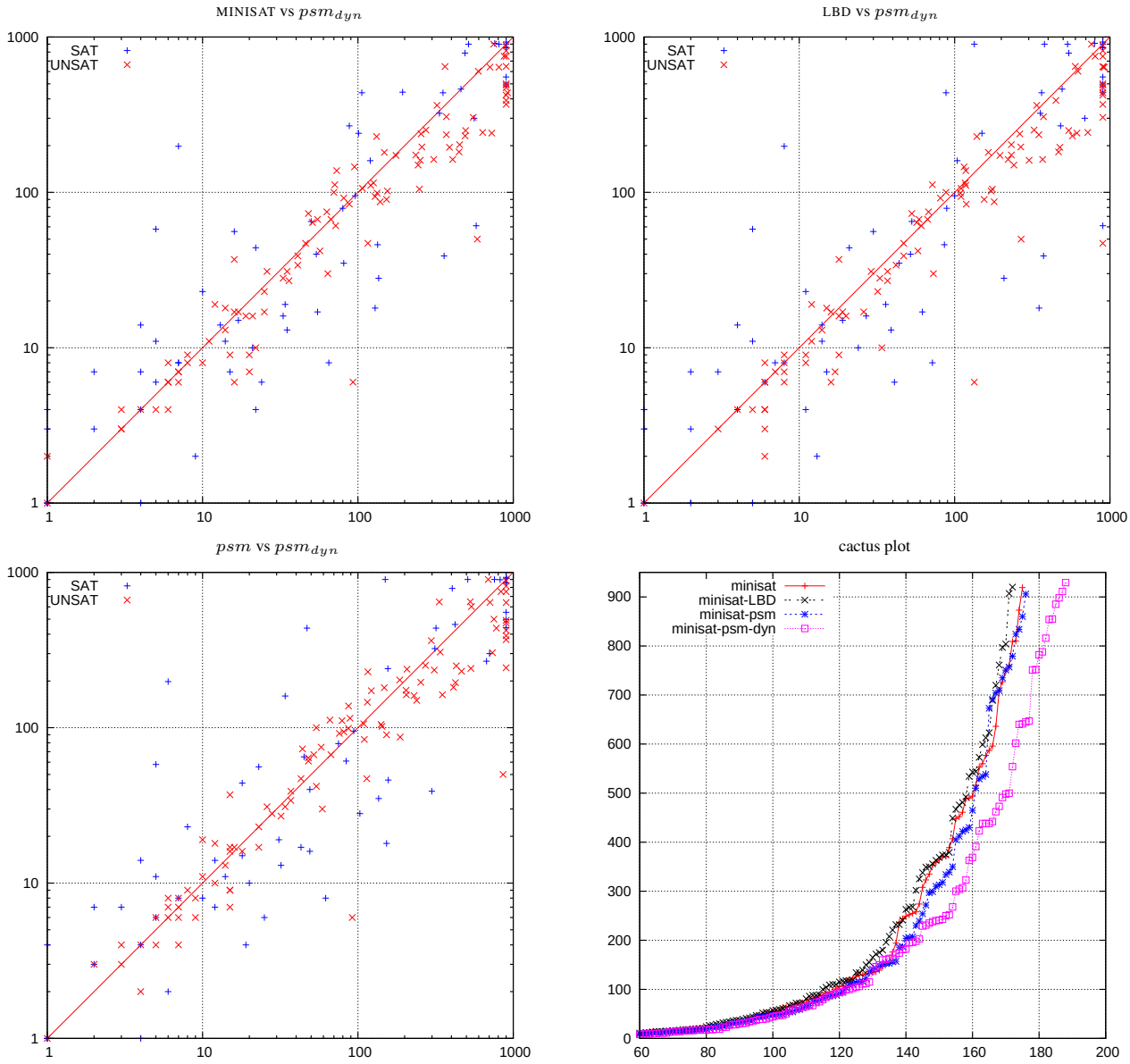


FIGURE 4 – Comparaison avec différentes stratégies de netoyage de la base de clauses apprises.

Pour l'ensemble des expérimentations, nous avons choisi d'utiliser MINISAT 2.2 comme solveur de base pour implémenter notre stratégie dynamique de réduction de la base de clauses apprises. Le solveur résultant est nommé MINISAT-psm_{dyn}. Le code source ainsi que des données additionnelles sont disponibles sur `http://www.cril.fr/~lagniez/ressource.html`.

5.1 Comparaison avec différentes stratégie de réduction

Nous comparons notre politique dynamique de nettoyage de la base de clauses apprises, appelée MINISAT-psm_{dyn}, avec MINISAT classique, et MINISAT avec la stratégie de réduction basée sur le *psm* (MINISAT-*psm*) et sur le LBD (MINISAT-LBD) (comme pour la section 3). La Figure 4 résume les résultats obtenus. Les trois nuages de points correspondent aux comparaisons de MINISAT-$psm_{psm_{dyn}}$ avec les trois autres solveurs. Chaque point correspond à une instance. Un point en dessous de la diagonale signifie que cette instance a été résolue plus rapidement avec MINISAT-$psm_{psm_{dyn}}$. Les instances SAT et UNSAT sont respectivement représentées sur la figure par le signe plus et multiplié. Cette figure contient aussi les courbes permettant de comparer le nombre d'instances résolues en fonction du temps pour les quatre solveurs.

Il est clair que notre approche de réduction de la base des clauses apprises permettant de geler des clauses est nettement meilleure que les trois autres. Le solveur résout 189 instances (76 SAT et 113 UNSAT), ce qui est meilleur que les autres solveurs (voir tableau 1). De plus, comme nous pouvons le voir sur le nuage de point, MINISAT-$psm_{psm_{dyn}}$

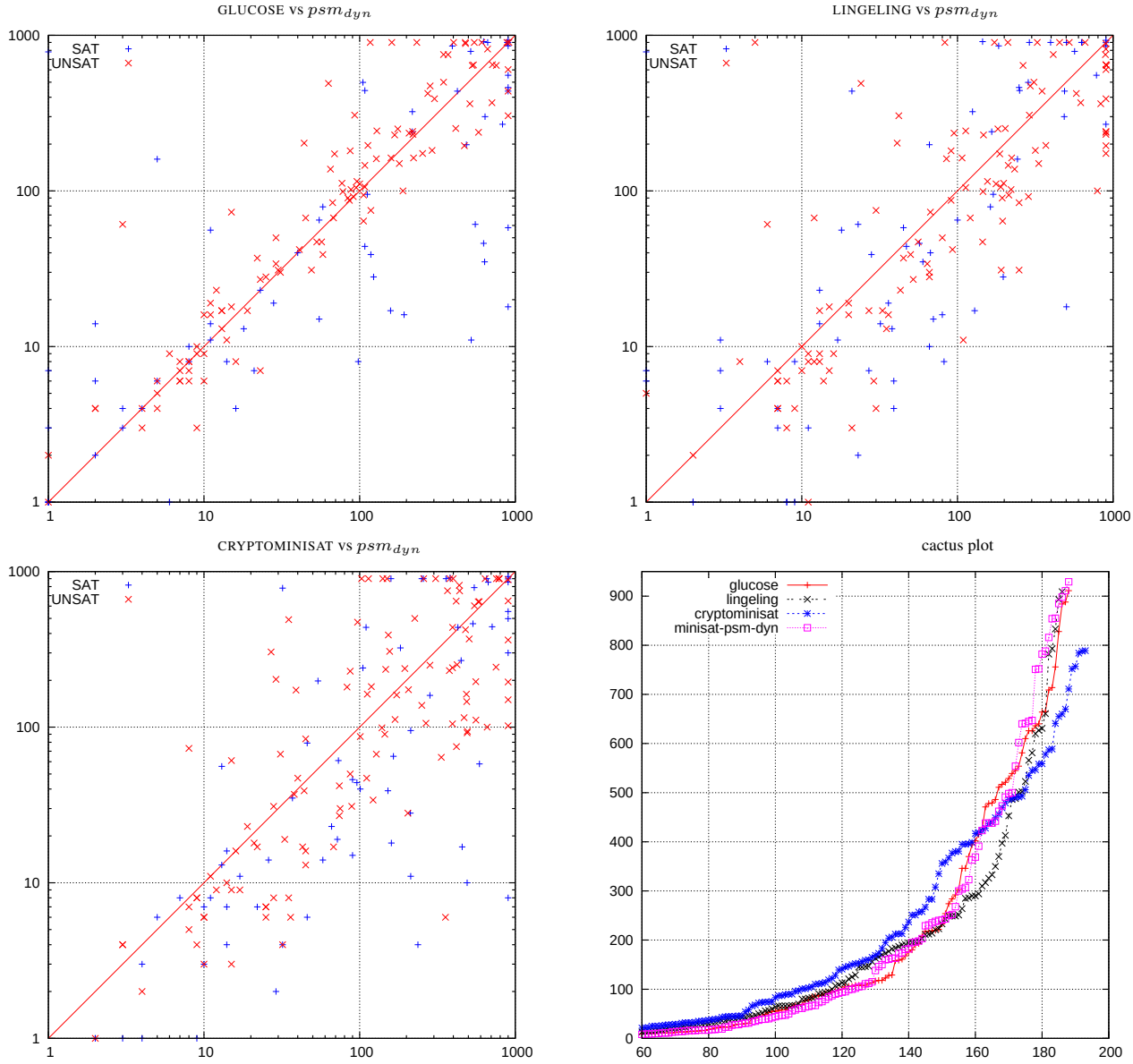


FIGURE 5 – Comparaison avec les solveurs de l'état de l'art : GLUCOSE, LINGELING and CRYPTOMINISAT.

résout les instances plus rapidement que les autres solveurs.

5.2 Comparaison avec les solveurs de l'état de l'art

Les résultats obtenus par les différents solveurs sont : LINGELING résout 187 instances (77 SAT et 110 UNSAT), GLUCOSE 189 instances (70 SAT et 119 UNSAT) et CRYPTOMINISAT 194 instances (74 SAT et 120 UNSAT). Donc notre approche basée sur la notion de réduction dynamique de la base de clauses apprises est compétitive avec les solveurs de l'état de l'art (rappelons qu'elle résout 189 instances (76 SAT et 113 UNSAT)). Ces résultats sont très encourageants puisque dernièrement (*ex* : redémarrages dynamiques, clauses bloquées, ...).

6 Conclusion

Dans ce papier, nous avons introduit une nouvelle mesure permettant d'identifier les clauses importantes pour la suite de la recherche. Cette mesure est dynamique (contrairement à la mesure basé sur le LBD) et peut être calculée sur des clauses ne participant pas à la recherche (contrairement à la mesure basée sur VSIDS). Grâce à ces propriétés, une nouvelle stratégie dynamique de réduction de la base de clauses apprises a été proposée. Cette nouvelle stratégie est basée sur l'activation et la désactivation de clauses, alors que avec les stratégies de réduction de l'état de l'art où les clauses sont définitivement supprimées. Un ensemble d'expérimentations a montré que cette approche a un très bon comportement.

En perspective, nous comptons étudier plus finement la

manière selon laquelle évolue l'interprétation $\mathcal{P}$, ce qui nous permettrait de décider quand une réduction de la base de connaissance serait souhaitable. De plus, comme la notion de *progress saving* a déjà été utilisée dans d'autres travaux [3, 14] nous pensons qu'il serait intéressant de combiner ces différentes approches.

Références

[1] G. Audemard and L. Simon. Predicting learnt clauses quality in modern SAT solvers. In *proceedings of International Joint Conference on Artificial Intelligence*, pages 399–404, 2009.

[2] Paul Beame, Henry Kautz, and Ashish Sabharwal. Towards understanding and harnessing the potential of clause learning. *Journal of Artificial Intelligence Research*, 22 :319–351, 2004.

[3] Armin Biere. Adaptive restart strategies for conflict driven SAT solvers. In *Theory and Applications of Satisfiability Testing - SAT 2008, 11th International Conference, SAT 2008, Guangzhou, China, May 12-15, 2008. Proceedings*, pages 28–33, 2008.

[4] Martin Davis, George Logemann, and Donald W. Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7) :394–397, 1962.

[5] N. Een and N. Sörensson. An extensible SAT-solver. In *proceedings of SAT*, pages 502–518, 2003.

[6] C. Gomes, B. Selman, and H. Kautz. Boosting combinatorial search through randomization. In *AAAI/IAAI*, pages 431–437, 1998.

[7] J. Huang. The effect of restarts on the efficiency of clause learning. In *proceedings of IJCAI*, pages 2318–2323, 2007.

[8] Matti Järvisalo, Armin Biere, and Marijn Heule. Blocked clause elimination. In *TACAS*, pages 129–144, 2010.

[9] Roberto J. Bayardo Jr. and Robert Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *AAAI/IAAI*, pages 203–208, 1997.

[10] J. Marques-Silva and K. Sakallah. GRASP - A New Search Algorithm for Satisfiability. In *proceedings of ICCAD*, pages 220–227, 1996.

[11] 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.

[12] Knot Pipatsrisawat and Adnan Darwiche. A lightweight component caching scheme for satisfiability solvers. In *SAT*, pages 294–299, 2007.

[13] Knot Pipatsrisawat and Adnan Darwiche. On the power of clause-learning sat solvers with restarts. In *cp09*, pages 654–668, 2009.

[14] Knot Pipatsrisawat and Adnan Darwiche. Width-based restart policies for clause-learning satisfiability solvers. In *12th International Conference on Theory and Applications of Satisfiability Testing - SAT 2009*, pages 341–355, 2009.

[15] L. Zhang, C. Madigan, M. Moskewicz, and S. Malik. Efficient conflict driven learning in boolean satisfiability solver. In *proceedings of ICCAD*, pages 279–285, 2001.

La contrainte d'équivalence dans QCSP(QBF)

Vincent Barichard, Igor Stéphan

LERIA, Université d'Angers, 2 Boulevard Lavoisier, 49045, Angers, Cedex 01, France
{barichard|stephan}@info.univ-angers.fr

Résumé

Le problème de validité d'une QBF dont la matrice propositionnelle est quelconque peut être vu comme un QCSP. Dans ce cadre, nous introduisons une nouvelle contrainte : la contrainte d'équivalence et prouvons qu'elle est plus puissante que la contrainte d'égalité. Pour illustrer sa pertinence, nous nous penchons sur QCSP(QBF) qui est un QCSP instancié au problème de validité des QBF avec formule propositionnelle quelconque. Nous rapportons un ensemble de résultats expérimentaux obtenus dans le cadre d'un environnement générique pour le développement de systèmes de contraintes.

Abstract

The QBF validity problem with any propositional formula may be seen as a QCSP. In this frame, we introduce a new constraint : the equivalence constraint and prove that it is stronger than equality constraint. To illustrate its relevance, we focus on QCSP(QBF) which is a QCSP instanciated to the QBF validity problem with any propositional formula. We report some experiments in a generic constraint development environment.

1 Introduction

Le formalisme des *Formules Booléennes Quantifiées* (ou QBF)[26] qui décrit une « hiérarchie » de langages a été introduit initialement comme un outil théorique dont le but était de décrire des classes de complexité au-delà de la classe NP. Mais depuis [8] qui s'est intéressé en premier au problème de validité des QBF prénexes sous forme normale conjonctive (ou PFNC), l'ensemble des questions liées à ce formalisme est aussi devenu un domaine de l'Intelligence Artificielle car elles ont de multiples applications : la planification dans l'incertain [20], en tant que langage cible pour l'implantation de différentes questions autour de formalismes logiques [5] ou en vérification formelle [4] pour ne citer que les plus connus. Ce formalisme est une restriction de la logique du second ordre mais peut être vu plus intuitivement comme une extension de la logique propositionnelle où les symboles propositionnels sont quantifiés ; une QBF est constituée d'une chaîne $q_1x_1 \dots q_nx_n$, le lieur (avec $x_1, \dots, x_n$ des symboles propositionnels distincts et $q_1, \dots, q_n$ des quantificateurs) et une formule propositionnelle, la matrice. Comme de coutume, $\exists$ représente le quantificateur existentiel et $\forall$ le quantificateur universel. Par exemple, $\forall a \forall b \exists c(\neg(a \leftrightarrow c) \rightarrow (b \leftrightarrow c))$ est une QBF ($\neg$ représente la négation, $\rightarrow$ l'implication, $\leftrightarrow$ la bi-implication et a, b et c sont des symboles propositionnels). La sémantique des connecteurs logiques est préservée et celle des quantificateurs s'inspire de celle de la logique du premier ordre (x est un symbole propositionnel, ϕ une QBF, $\vee$ représente la disjonction, $\wedge$ représente la conjonction, les constantes propositionnelles $\top$ et $\bot$ représentent respectivement ce qui est toujours vrai et ce qui est toujours faux, $[\ \leftarrow\ ]$ représente la substitution) :

$$\exists x\phi = ([x \leftarrow \bot](\phi) \vee [x \leftarrow \top](\phi))$$

et

$$\forall x\phi = ([x \leftarrow \bot](\phi) \wedge [x \leftarrow \top](\phi)).$$

Une QBF est valide si $\phi \equiv \top$ (avec le symbole « $\equiv$ » pour l'équivalence logique). Par exemple, la QBF $\forall z \exists y \exists x((x \vee y) \leftrightarrow z)$ est valide tandis que la QBF $\exists y \exists x \forall z((x \vee y) \leftrightarrow z)$ ne l'est pas. Le problème de satisfiabilité d'une formule propositionnelle (SAT) n'est autre que le problème de validité d'une QBF dont un lieur est uniquement constitué de quantificateurs existentiels. Ainsi, la plus grande part des procédures de décisions récentes pour le problème de validité des QBF (cf. [14]) sont basées sur l'algorithme de recherche dit de « Davis, Logemann et Loveland » [9] (ou DLL) pour SAT qui est une conséquence directe de la sémantique des quantificateurs existentiels. (D'autres procédures de décisions sont basées sur le principe de résolution telle que la Q-résolution [16] ou Quantor [6] ;

d'autres, tels QSAT [19] ou QMRES [18] sont basées sur un algorithme d'élimination des quantificateurs à la Fourier-Motzkin ; d'autres enfin, tel skizzo [3] sont basées sur la skolémisation).

La forme normale conjonctive (ou FNC) est un fragment privilégié pour SAT et les algorithmes de recherche car on peut détecter localement certaines formes d'insatisfiabilité. Puisque la plus grande part des procédures de décision pour QBF basées sur le DLL sont des extensions de procédures SAT [20, 12], elles sont aussi basées sur les QBF FNC. Mais ce fragment n'est pas adapté aux QBF car il ne reflète pas la dualité de la sémantique des quantificateurs qui est une conséquence directe de celle de la sémantique de la conjonction et de la disjonction. Ainsi, une forme mixte entre FND (forme normale disjonctive) et FNC [28, 21], une forme normale négative (FNN) [2, 17, 10], voire un fragment moins strict (par exemple [15] qui accepte toute formule sur $\{\neg, \wedge, \vee\}$) sont utilisés en entrée des procédures de décision modernes pour le problème de validité des QBF.

Si une QBF quelconque est considérée, par introduction de nouveaux symboles propositionnels [7] à la Tseitin [27] (voire de littéraux [24]), le problème de validité peut être vu comme un problème de *satisfaction de contraintes quantifiées* (QCSP) [11]. Par exemple, $\forall a \forall b \exists c(((a \rightarrow c) \rightarrow \neg(c \rightarrow a)) \rightarrow \neg((b \rightarrow c) \rightarrow \neg(c \rightarrow b)))$ est valide si et seulement si la décomposition suivante de la QBF est valide [1]

$$\begin{array}{ll} & \forall a \forall b \exists c \exists o_0 \exists o_1 \exists o_2 \exists o_3 \exists o_4 \exists o_5 \\ & \quad ((o_2 \rightarrow \overline{o_5}) \leftrightarrow \top) \wedge ((o_0 \rightarrow \overline{o_1}) \leftrightarrow o_2) \\ \wedge & \quad ((a \rightarrow c) \leftrightarrow o_0) \wedge ((c \rightarrow a) \leftrightarrow o_1) \\ \wedge & \quad ((o_3 \rightarrow \overline{o_4}) \leftrightarrow o_5) \wedge ((b \rightarrow c) \leftrightarrow o_3) \\ \wedge & \quad ((c \rightarrow b) \leftrightarrow o_4) \end{array}$$

seulement si chaque QBF (les *contraintes*, notées c_i) dans l'ensemble ci-dessous le sont aussi :

$$\left\{ \begin{array}{ll} c_1 : & \exists o_2 \exists o_5((o_2 \rightarrow \overline{o_5}) \leftrightarrow \top), \\ c_2 : & \exists o_0 \exists o_1 \exists o_2((o_0 \rightarrow \overline{o_1}) \leftrightarrow o_2), \\ c_3 : & \forall a \exists c \exists o_0((a \rightarrow c) \leftrightarrow o_0), \\ c_4 : & \forall a \exists c \exists o_1((c \rightarrow a) \leftrightarrow o_1), \\ c_5 : & \exists o_3 \exists o_4 \exists o_5((o_3 \rightarrow \overline{o_4}) \leftrightarrow o_5), \\ c_6 : & \forall b \exists c \exists o_3((b \rightarrow c) \leftrightarrow o_3), \\ c_7 : & \forall b \exists c \exists o_4((c \rightarrow b) \leftrightarrow o_4) \end{array} \right\}$$

La figure 1 montre des arbres de recherche dans lesquels chaque nœud (sauf la racine) est étiqueté par une substitution sur $\{\top, \bot\}$ et chaque arc est étiqueté par une propagation ; dans les deux arbres, les feuilles sont étiquetées par *succès* ou *échec*. Nous déroulons dans le premier temps une branche de l'arbre de recherche pour la figure 1.1. Si a est substitué par $\top$ alors, par propagation, le domaine booléen de o_1 est restreint aussi à $\top$ par la contrainte c_4, et selon la définition des CSP [1], une contrainte d'égalité entre o_0 et c par la contrainte c_3 ainsi qu'une contrainte d'inégalité entre o_2 et o_0 par la contrainte c_2 peuvent être ajoutées à l'ensemble des contraintes ; ce choix sur a et ses conséquences sont étiquetés par (1) dans l'arbre ainsi que dans le tableau des contraintes d'égalité et d'inégalité. Selon la sémantique, un échec revient sur le dernier choix fait pour un symbole existentiellement quantifié tandis qu'un succès revient sur le dernier choix fait pour un symbole universellement quantifié (s'il n'y en a plus, c'est un succès global). Poursuivant notre exemple, si b est substitué par $\top$ alors le domaine booléen de o_4 est restreint à $\top$ par la contrainte c_7, et une contrainte d'égalité entre o_3 et c par la contrainte c_6 ainsi qu'une contrainte d'inégalité entre o_5 et o_3 par la contrainte c_5 peuvent être ajoutées à l'ensemble des contraintes ; ceci étant étiqueté (2). Enfin, si c est substitué par $\bot$ alors le domaine de o_3 est restreint à $\bot$ par la contrainte c_6, le domaine de o_5 est restreint à $\top$ par la contrainte c_5, le domaine de o_0 est restreint à $\bot$ par la contrainte c_3 et le domaine de o_2 est restreint à $\top$ par la contrainte c_2 : ceci est consistant avec les contraintes d'égalités et d'inégalités mais inconsistant avec la contrainte c_1 (d'où l'échec) ; ceci étant étiqueté (3). La figure 1.2 montre aussi l'arbre de recherche mais pour un mécanisme de propagation qui prend en compte l'équivalence. Revenant sur le choix de a à $\top$, selon les propriétés booléennes, le domaine de o_1 est toujours restreint à $\top$ par la contrainte c_4, mais les deux équivalences $(o_0 \leftrightarrow c)$ et $(\overline{o_0} \leftrightarrow o_2)$ (et donc $(o_2 \leftrightarrow \overline{c})$ par transitivité) peuvent être considérées ; ceci étant étiqueté (4). En poursuivant l'exemple, si b est substitué par $\top$ alors le domaine de o_4 est toujours restreint à $\top$ par la contrainte c_7 et deux équivalences $(o_3 \leftrightarrow c)$ et $(\overline{o_3} \leftrightarrow o_5)$ (et donc $(o_5 \leftrightarrow \overline{c})$ par transitivité) peuvent être considérées. Bien que les domaines des symboles propositionnels o_2 et o_5 de la contrainte c_1 : $((o_2 \rightarrow \overline{o_5}) \leftrightarrow \top)$ n'aient pas changé, par l'application des équivalences $(o_2 \leftrightarrow \overline{c})$ et $(o_5 \leftrightarrow \overline{c})$, c est restreint par propagation à $\top$ (étant donné que c_1 est par application des équivalences $((\overline{c} \rightarrow c) \leftrightarrow \top)$) ; ceci étant étiqueté (5). Dans l'arbre de recherche par propagation sans équivalence, le nombre de nœuds explorés est de 11 tandis qu'il n'est que de 6 si nous prenons en compte l'équivalence. De plus, l'échec a été évité. Cet exemple démontre que la propagation prenant en compte une contrainte d'équivalence est plus puissante que la propagation sans équivalence.

Un CSP peut être conçu pour manipuler différents domaines (booléen, entier, flottant). Le terme

1. l'opérateur de complémentation $\overline{(.)}$ est tel que si x est un symbole propositionnel alors $\overline{x} = \neg x$, en particulier $\overline{\overline{x}} = x$; x et $\overline{x}$ sont des littéraux.

FIGURE 1 – Arbres de recherche

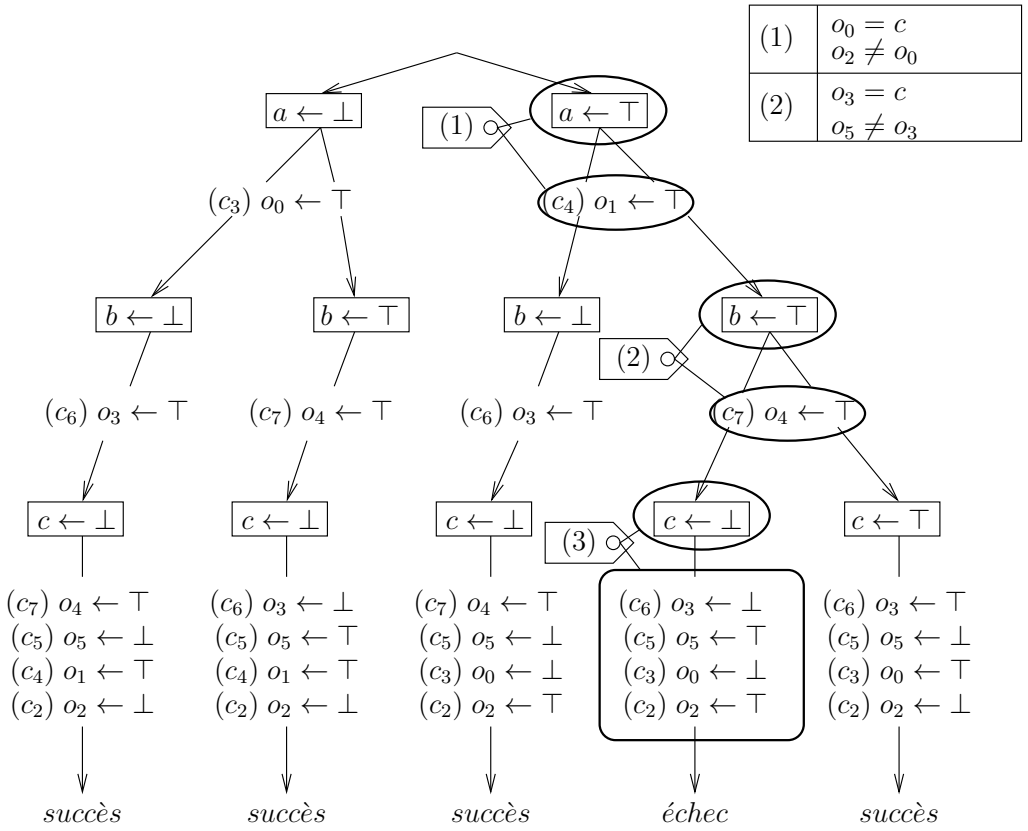


Figure 1.1 - Arbre de recherche *sans* contrainte d'équivalence

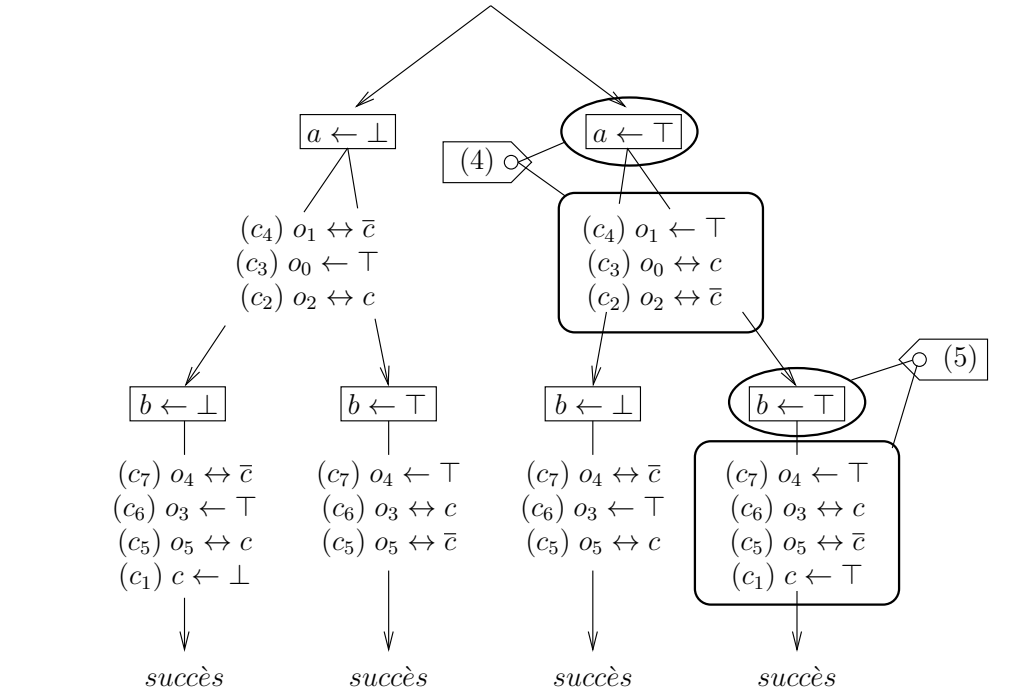


Figure 1.2 - Arbre de recherche *avec* contrainte d'équivalence

CSP(Bool) peut être employé pour dénommer les travaux autour du problème de satisfiabilité des formules booléennes. De même, nous utilisons le terme QCSP(Bool) pour caractériser les travaux autour du problème de validité des QBF. Dans ces derniers, les propagateurs maintiennent une consistance locale sur les domaines booléens ; les quantificateurs n'étant pris en charge que par l'algorithme de recherche. Pour améliorer la propagation, il a été proposé d'utiliser les propriétés spécifiques des QBF pour concevoir de nouveaux propagateurs [7] : nous nommons ces travaux QCSP(QBF). Pour aller encore plus loin, nous proposons d'ajouter la contrainte d'équivalence aux QCSP(QBF).

Une manière d'appliquer l'équivalence, en respectant l'ordre induit par le lieur, est de remplacer le symbole propositionnel par son équivalent soit par référence [24] soit par une réécriture des contraintes : dans les deux cas, nous nous situons hors du cadre classique des QCSP [11].

Dans le but de présenter notre nouvelle contrainte d'équivalence, nous faisons quelques rappels dans la section suivante sur le système de séquents S_{QBF} [25] qui la sous-tend. En nous basant sur ce système, nous utilisons un solveur QCSP qui est un solveur CSP instancié avec un algorithme de recherche quantifié et étendu pour prendre en compte un nouveau type de domaine : le domaine d'équivalence. Pour maintenir notre nouvelle contrainte, nous avons développé de nouveaux propagateurs basés sur notre nouveau domaine. Nous reportons quelques expériences dans le cadre de l'environnement générique pour le développement de systèmes de contraintes Gecode [22].

2 Préliminaire de théorie de la preuve

L'introduction de la contrainte d'équivalence s'appuie sur le calcul syntaxique S_{QBF} présenté en détail dans [25]. Ce système est basé sur la notion de relation de formules [23] (propositionnelle). Une relation de formule R pour une formule F est définie comme étant une relation d'équivalence sur l'ensemble des sous-formules avec pour contrainte que pour toute sous-formule A et B 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). Ce formalisme est étendu aux QBF QM en ajoutant un lieur à la partition P des sous-formules de 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 des sous-formules de 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 $([A]_R = [B]_R)$.

Le système syntaxique S_{QBF} pour les QBF est 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 $Id_{QM}([M] = [\top])$. 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 est plus à gauche que u dans le lieur.

Les règles du système S_{QBF} ne sont présentées 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).

Le système S_{QBF} est constitué de deux règles d'élimination du quantificateur existentiel :

$$\exists\top : \frac{(Q;[x]=[\top])}{(\exists xQ;[x])} \qquad \exists\bot : \frac{(Q;[\overline{x}]=[\top])}{(\exists xQ;[x])}$$

d'une règle d'élimination du quantificateur universel :

$$\forall : \frac{(Q;[\overline{x}]=[\top]) \quad (Q;[x]=[\top])}{(\forall xQ;[x])}$$

et neuf règles de fusion des classes :

$$1 : \frac{(Q;[\overline{F_X}]=[\top],[\overline{F_Y}]=[\top])}{(Q;[(F_X \rightarrow F_Y)]=[\overline{F_Y}])} \qquad 2 : \frac{(Q;[F_X]=[\top],[F_Y]=[\top])}{(Q;[(F_X \rightarrow F_Y)]=[F_X])}$$

$$3 : \frac{(Q;[F_X]=[\top],[\overline{F_Y}]=[\top])}{(Q;[(F_X \rightarrow F_Y)]=[\top])} \qquad 4 : \frac{(Q;[(F_X \rightarrow F_Y)]=[\overline{F_X}])}{(Q;[(F_X \rightarrow F_Y)],[\overline{F_Y}]=[\top])}$$

$$5 : \frac{(Q;[(F_X \rightarrow F_Y)]=[\overline{F_X}])}{(Q;[(F_X \rightarrow F_Y)],[F_X]=[\overline{F_Y}])} \qquad 6 : \frac{(Q;[(F_X \rightarrow F_Y)]=[\top])}{(Q;[(F_X \rightarrow F_Y)],[\overline{F_X}]=[\top])}$$

$$7 : \frac{(Q;[(F_X \rightarrow F_Y)]=[\top])}{(Q;[(F_X \rightarrow F_Y)],[F_Y]=[\top])} \qquad 8 : \frac{(Q;[(F_X \rightarrow F_Y)]=[\top])}{(Q;[(F_X \rightarrow F_Y)],[F_X]=[F_Y])}$$

$$9 : \frac{(Q;[(F_X \rightarrow F_Y)]=[F_Y])}{(Q;[(F_X \rightarrow F_Y)],[F_X]=[\top])}$$

Un *arbre de déduction* est défini inductivement, chaque nœud correspondant à une relation de formule non explicitement contradictoire. L'arbre croît à l'aide des règles du système S_{QBF}.

Un *axiome* est une relation de formules

1. non explicitement contradictoire ne contenant que deux classes et

2. 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.

La définition de l'axiome prévient la contradiction dans une classe. Cette définition se réduit uniquement à la condition 1 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 auraient déjà déduit la contradiction) ce qui sera le cas dans ce qui suit.

Le système S_{QBF} est correct et complet vis-à-vis de la sémantique des QBF [25].

La figure 2 développe l'arbre de preuve pour l'exemple introductif.

3 De QBF vers QCSP(QBF)

Dans le but de définir notre QCSP(QBF), dont le système S_{QBF} est la justification dans la théorie de la preuve, comme un CSP associé à un algorithme de recherche quantifié, nous avons besoin de définir un nouveau domaine pour les variables constitué d'un domaine booléen et un *domaine d'équivalence* avec des opérations qui respectent les propriétés liées à l'ordre des quantificateurs et de l'équivalence.

Définition 1 (domaine d'équivalence)
L'ensemble des polarités ordonnées est l'ensemble $\mathbb{N}^{+-} = \{i^-, i^+ | i \in \mathbb{N}^*\}$.

- *Une opération de complémentation* $\overline{(.)}$ *sur* $\mathbb{N}^{+-}$ *est définie par* $\overline{i^+} = i^-$ *et* $\overline{i^-} = i^+$ *pour tout* $i \in \mathbb{N}^*$.
- *Une relation d'ordre* $\prec$ *sur* $\mathbb{N}^{+-}$ *est définie ainsi : pour tout* $i \in \mathbb{N}^*$, $i^- \prec (i+1)^-$, $i^- \prec (i+1)^+$, $i^+ \prec (i+1)^-$ *et* $i^+ \prec (i+1)^+$.
- *Une fonction* $[.]$ *de* $\mathbb{N}^{+-}$ *dans* $2^{\mathbb{N}^{+-}}$ *est définie ainsi : pour tout* $i \in \mathbb{N}^*$, $[i^-] = \{1^-, 1^+, \ldots, (i-1)^-, (i-1)^+, i^-\}$ *et* $[i^+] = \{1^-, 1^+, \ldots, (i-1)^-, (i-1)^+, i^+\}$.
- *Une fonction constDomEq de* $\{\exists, \forall\} \times \mathbb{N}^*$ *dans* $2^{\mathbb{N}^{+-}}$ *est définie ainsi :*

  si $q = \exists$ ***alors***
  $constDomEq(q, i) = [i^+]$
  sinon
  $constDomEq(q, i) = \{i^+\}$
  fin si

L'ensemble des domaines d'équivalence est l'ensemble $ED = \{[i^-], [i^+], \{i^-\}, \{i^+\} | i \in \mathbb{N}^*\}$. *Une opération de complémentation* $\overline{(.)}$ *sur* ED *est définie ainsi : pour tout* $d \in ED$, $\overline{d} = (d \setminus \max(d)) \cup \{\overline{\max(d)}\}$.

L'intuition du domaine d'équivalence pour une variable est comment celle-ci est connectée à sa classe et avec quelle polarité. Pour l'exemple introductif, les domaines d'équivalence initiaux des variables sont :

$$\begin{array}{l}\mathrm{DomEq}(a) = \{1^+\}, \mathrm{DomEq}(b) = \{2^+\},\\ \mathrm{DomEq}(c) = [3^+], \mathrm{DomEq}(o0) = [4^+],\\ \mathrm{DomEq}(o1) = [5^+], \mathrm{DomEq}(o3) = [6^+],\\ \mathrm{DomEq}(o4) = [7^+], \mathrm{DomEq}(o2) = [8^+],\\ \mathrm{DomEq}(o5) = [9^+].\end{array}$$

Définition 2 (relations et opérateurs de filtrage du CSP(QBF)) *Le domaine d'une variable* v *est une paire* $(DomBool(v), DomEq(v))$ *dans* $2^{\{\top,\bot\}} \times ED$. *La table 1 définit quatre relations unaires* $(= \top)$, $(= \bot)$, $(\neq \top)$ *et* $(\neq \bot)$, *deux relations binaires* $\sim$ *et* $\not\sim$ *et une relation ternaire imp avec leurs opérateurs de filtrage respectifs.*

Les opérateurs de filtrage maintiennent un domaine d'équivalence de telle manière que si les conditions de l'explicitement contradictoire sont vérifiées alors celui-ci devient vide. Ces opérateurs de filtrage appliquent également les neuf règles de fusion du système S_{QBF}.

Nous définissons maintenant notre QCSP(QBF).

Définition 3 (QCSP(QBF)) *Un QCSP(QBF) est un quintuplet* $\langle \mathbf{V}, ordre, quant, \mathbf{D}, \mathbf{C} \rangle$ *avec*

$$\mathbf{D} = \{(\{\bot, \top\}, constDomEq(quant(v), ordre(v))) | v \in \mathbf{V}\}$$

Chaque symbole propositionnel de la QBF est associé à une variable et $\mathbf{C}$ *est un ensemble de contraintes sur les variables de* $\mathbf{V}$ *des relations* $(= \top)$, $(= \bot)$, $(\neq \top)$, $(\neq \bot)$, $\sim$, $\not\sim$ *et imp.*

Nous illustrons sur quelques exemples comment les domaines d'équivalence sont mises à jour lorsque les opérateurs de filtrage sont appliqués. Soit le lieur $\forall a \forall b \exists c \exists d$ et les domaines d'équivalence associés : $\{1^+\}, \{2^+\}, [3^+]$ et $[4^+]$. Soit la formule $(a \leftrightarrow b)$, en appliquant l'opérateur $(a \sim b)$ (cf table 1) nous obtenons deux nouveaux domaines $\mathrm{DomEq}(a) = \mathrm{DomEq}(b) = \emptyset$, un échec est constaté et la propagation est interrompue. Nous obtenons le même résultat si une variable exisentiellement quantifiée est mise en équivalence avec une variable universellement quantifiée telles que la première précède la seconde dans le lieur. Maintenant considérons la formule $((d \leftrightarrow \overline{c}) \wedge (c \leftrightarrow a))$. Les deux opérateurs de filtrage impliqués sont $(d \not\sim c)$ et $(c \sim a)$. Nous appliquons en premier la contrainte $(d \not\sim c)$ et mettons à jour le domaine d'équivalence de la variable d à $[3^-]$; alors nous appliquons l'opérateur $(c \sim a)$ et mettons à jour le domaine d'équivalence de c à jour à $\{1^+\}$; puisque le domaine de la variable c a changé, la propagation réveille la contrainte $(d \not\sim c)$ et mettons à jour le domaine d'équivalence de la variable d à

FIGURE 2 – Arbre de preuve pour la QBF $\forall a \forall b \exists c M$ avec $M = (((a \to c) \to \neg(c \to a)) \to \neg((b \to c) \to \neg(c \to b)))$, $F_0 = (a \to c)$, $F_1 = (c \to a)$, $F_4 = (c \to b)$, $F_3 = (b \to c)$, $F_2 = (F_0 \to \overline{F_1})$, $F_5 = (F_3 \to \overline{F_4})$ et $M = (F_2 \to \overline{F_5})$. Les classes fusionnées apparaissent dans la partie supérieure du séquent.

$$\dfrac{\dfrac{\dfrac{\dfrac{\nabla_\bot}{(\forall b \exists c; [M, \top, \overline{a}] = [F_0], [F_1, \overline{c}])}\,9}{(\forall b \exists c; [M, \top, \overline{a}], [F_1] = [\overline{c}])}\,6}{(\forall b \exists c; [M, \top] = [\overline{a}])}\,4 \qquad \dfrac{\dfrac{\dfrac{\nabla_\top}{(\forall b \exists c; [M, \top, a, F_1], [F_0] = [c])}\,4}{(\forall b \exists c; [M, \top, a] = [F_1])}\,9}{(\forall b \exists c; [M, \top] = [a])}\,7}{(\forall a \forall b \exists c; [M] = [\top])}\,\forall$$

avec $\nabla_\bot$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{(\varepsilon; [F_3, M, \top, \overline{a}, F_0, \overline{b}, \overline{F_5}, F_4, F_1, \overline{c}, \overline{F_2}])}{(\exists c; [F_3, M, \top, \overline{a}, F_0, \overline{b}] = [\overline{F_5}, F_4, F_1, \overline{c}, \overline{F_2}])}\,\exists\bot}{(\exists c; [F_3, M, \top, \overline{a}, F_0, \overline{b}], [\overline{F_5}] = [F_4, F_1, \overline{c}, \overline{F_2}])}\,8}{(\exists c; [F_3] = [M, \top, \overline{a}, F_0, \overline{b}], [F_4, F_1, \overline{c}, \overline{F_2}])}\,9}{(\exists c; [M, \top, \overline{a}, F_0, \overline{b}], [F_4] = [F_1, \overline{c}, \overline{F_2}])}\,6}{(\exists c; [M, \top, \overline{a}, F_0] = [\overline{b}], [F_1, \overline{c}, \overline{F_2}])}\,4 \qquad \dfrac{\dfrac{\dfrac{\dfrac{(\exists c; [M, \top, \overline{a}, F_0, b, F_4], [F_1, \overline{c}, \overline{F_2}, \overline{F_3}, F_5])}{(\exists c; [M, \top, \overline{a}, F_0, b, F_4], [F_1, \overline{c}, \overline{F_2}, \overline{F_3}] = [F_5])}\,8}{(\exists c; [M, \top, \overline{a}, F_0, b, F_4], [F_1, \overline{c}, \overline{F_2}] = [\overline{F_3}])}\,4}{(\exists c; [M, \top, \overline{a}, F_0, b] = [F_4], [F_1, \overline{c}, \overline{F_2}])}\,9}{(\exists c; [M, \top, \overline{a}, F_0] = [b], [F_1, \overline{c}, \overline{F_2}])}\,7}{(\forall b \exists c; [M, \top, \overline{a}, F_0], [F_1, \overline{c}] = [\overline{F_2}])}\,\forall$$

et $\nabla_\top$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{(\exists c; [M, \top, a, F_1, \overline{b}, F_3], [\overline{F_0}, \overline{c}, F_2, F_4, \overline{F_5}])}{(\exists c; [M, \top, a, F_1, \overline{b}, F_3], [F_0, c, \overline{F_2}, \overline{F_4}] = [F_5])}\,8}{(\exists c; [M, \top, a, F_1, \overline{b}] = [F_3], [F_0, c, \overline{F_2}, \overline{F_4}])}\,9}{(\exists c; [M, \top, a, F_1, \overline{b}], [F_0, c, \overline{F_2}] = [\overline{F_4}])}\,6}{(\exists c; [M, \top, a, F_1, \overline{b}], [F_0, c, \overline{F_2}])}\,4 \qquad \dfrac{\dfrac{\dfrac{\dfrac{(\varepsilon; [M, \top, a, F_1, b, F_4, F_0, c, \overline{F_2}, F_3, \overline{F_5}])}{(\exists c; [M, \top, a, F_1, b, F_4] = [F_0, c, \overline{F_2}, F_3, \overline{F_5}])}\,\exists\top}{(\exists c; [M, \top, a, F_1, b, F_4], [F_0, c, \overline{F_2}, F_3] = [\overline{F_5}])}\,8}{(\exists c; [M, \top, a, F_1, b, F_4], [F_0, c, \overline{F_2}] = [F_3])}\,4}{(\exists c; [M, \top, a, F_1, b] = [F_4], [F_0, c, \overline{F_2}])}\,9}{(\exists c; [M, \top, a, F_1, b], [F_0, c, \overline{F_2}])}\,7}{(\forall b \exists c; [M, \top, a, F_1], [F_0, c] = [\overline{F_2}])}\,\forall$$

TABLE 1 – Relations et opérateurs de filtrage pour CSP(QBF)

$v = \top$, v une variable	$\mathrm{DomBool}(v) = \{\top\}$
$v = \bot$, v une variable	$\mathrm{DomBool}(v) = \{\bot\}$
$v \neq \top$, v une variable	$\mathrm{DomBool}(v) = \{\bot\}$
$v \neq \bot$, v une variable	$\mathrm{DomBool}(v) = \{\top\}$
$v_i \sim v_j$, v_i, v_j deux variables	**si** $\overline{\mathrm{DomEq}(v_i)} = \mathrm{DomEq}(v_j)$ ou $\mathrm{DomEq}(v_i) \cap \mathrm{DomEq}(v_j) = \emptyset$ **alors** $\mathrm{DomEq}(v_i) = \mathrm{DomEq}(v_j) = \emptyset$ **sinon si** $\max(\mathrm{DomEq}(v_i)) \prec \max(\mathrm{DomEq}(v_j))$ **alors** $\mathrm{DomEq}(v_j) = \mathrm{DomEq}(v_i)$ **sinon** $\mathrm{DomEq}(v_i) = \mathrm{DomEq}(v_j)$ **fin si**
$v_i \not\sim v_j$, v_i, v_j deux variables	**si** $\mathrm{DomEq}(v_i) = \mathrm{DomEq}(v_j)$ ou $\overline{\mathrm{DomEq}(v_i)} \cap \mathrm{DomEq}(v_j) = \emptyset$ **alors** $\mathrm{DomEq}(v_i) = \mathrm{DomEq}(v_j) = \emptyset$ **sinon si** $\max(\mathrm{DomEq}(v_i)) \prec \max(\mathrm{DomEq}(v_j))$ **alors** $\mathrm{DomEq}(v_j) = \overline{\mathrm{DomEq}(v_i)}$ **sinon** $\mathrm{DomEq}(v_i) = \overline{\mathrm{DomEq}(v_j)}$ **fin si**
$\mathrm{imp}(v_i, v_j, v_k)$, v_i, v_j, v_k des variables ou $\top$ ou $\bot$	**cas** $v_k \not\sim v_j : v_i \neq \top, v_j \neq \top$ **cas** $v_i \sim v_k : v_i = \top, v_j = \top$ **cas** $v_k = \bot : v_i = \top, v_j \neq \top$ **cas** $v_j = \bot : v_k \not\sim v_i$ **cas** $v_i \not\sim v_j : v_k \not\sim v_i$ **cas** $v_i = \bot : v_k = \top$ **cas** $v_j = \top : v_k = \top$ **cas** $v_i \sim v_j : v_k = \top$ **cas** $v_i = \top : v_k \sim v_j$

$\{1^-\}$. Ainsi à la fin de la propagation, nous obtenons trois domaines d'équivalences : DomEq(a) = $\{1^+\}$, DomEq(c) = $\{1^+\}$ et DomEq(d) = $\{1^-\}$.

La propagation seule n'est pas suffisante pour résoudre un QCSP. Pour ce faire, nous avons besoin d'un algorithme de recherche quantifié pour atteindre la complétude vis-à-vis du QCSP [11].

Définition 4 (algorithme de recherche quantifié $QCSP\_QBF$)

Entrée: *Une QBF ϕ*
Sortie: `valide` *si la QBF ϕ est valide et* `non_valide` *sinon*
C := *décompose*(ϕ)
tant que *vrai* ***faire***
 selon *propage*(C) ***faire***
  cas *échec*
  retour-arrière jusqu'au dernier choix sur un symbole propositionnel existentiellement quantifié
  si *aucun* ***alors***
   retourner `non_valide`
  fin si
  cas *succès*
  retour-arrière jusqu'au dernier choix sur un symbole propositionnel universellement quantifié
  si *aucun* ***alors***
   retourner `valide`
  fin si
  cas *branche*
  sélectionne le symbole propositionnel suivant x
  sélectionne une valeur booléenne K pour le symbole propositionnel
  ajoute à C la contrainte $(x = K)$
 fin selon
fin tant que

Dans la définition 4, la fonction *décompose* fait référence à la décomposition d'une formule en contraintes selon l'introduction de littéraux existentiellement quantifiés [24] et la fonction *propage* propage toute modification des domaines jusqu'à ce qu'un point-fixe soit atteint.

Les valeurs des nouveaux domaines accolés aux domaines booléens ne sont pas utilisées lors des branchements, ils n'augmentent donc pas la taille de l'arbre de recherche et sont seulement utilisés lors des phases de propagations.

Les règles d'élimination du système S_{QBF} sont assurées par l'algorithme de recherche quantifié. Le calcul de point-fixe de la propagation permet de définir l'axiome du système de séquents uniquement par sa condition 1 (i.e. la mise à jour d'une relation de formules explicitement contradictoire).

Le théorème suivant fait le lien entre l'algorithme $QCSP\_QBF$ et le système S_{QBF}.

Théoreme 1 (Correction et complétude de l'algorithme $QCSP\_QBF$ vis-à-vis du système S_{QBF}) *Soit une QBF QM. $Id_{QM}([M] = [\top])$ admet une preuve si et seulement si $QCSP\_QBF(QM)$ retourne* `valide`*. $Id_{QM}([M] = [\top])$ n'admet aucune preuve si et seulement si $QCSP\_QBF(QM)$ retourne* `non_valide`*.*

Preuve esquissée du théorème 1. Le domaine d'équivalence et les opérateurs de filtrage associés assurent la notion d'explicitement contradictoire : un domaine d'équivalence pour une variable ne peut contenir qu'une seule polarité (ou être vide) ; quand un domaine d'équivalence pour une variable universellement quantifiée est mise à jour avec le domaine d'équivalence d'une autre variable universellement quantifiée alors les deux domaines deviennent vide car l'intersection l'est ; quand un domaine d'équivalence pour une variable existentiellement quantifiée est mise à jour avec le domaine d'équivalence d'une variable universellement quantifiée telle que la première précède la seconde dans le lieur alors les deux domaines deviennent vide car l'intersection l'est. L'algorithme de recherche quantifiée implémente le système S_{QBF}. Chaque exécution de l'algorithme peut être simulée par une preuve dans le système S_{QBF}. Réciproquement, de chaque preuve dans le système S_{QBF} il est possible de construire un arbre de recherche pour l'algorithme de recherche quantifié ; celui-ci étant construit récursivement par permutation des règles de fusion et montée dans l'arbre des règles d'élimination des quantificateurs.

4 Résultats expérimentaux

Nous avons implanté la contrainte d'équivalence dans Gecode [22], un environnement générique de développement de contraintes (i.e. une boîte à outils pour développer des systèmes à base de contraintes) implémenté en C++. Nous n'avons en rien eu besoin de modifier Gecode. Nous avons implémenté le domaine d'équivalence avec des intervalles et déconnecté toutes les optimisations. La table 2 rapporte quelques résultats expérimentaux. La première colonne donne le nom du benchmark issu de la bibliothèque $QBFLIB\_1.0$ [13] ; la seconde colonne montre le lieur du benchmark ; les troisième, quatrième et cinquième colonnes rapportent respectivement le nombre de propagations, le nombre de nœuds de l'arbre de recherche et le nombre d'échecs pour le solveur *sans* la contrainte d'équivalence ; la sixième colonne montre le gain en temps de calcul ($= \frac{\text{le temps cpu sans*100}}{\text{le temps cpu avec}} - 100$, ∞ signifie que le solveur n'a pas été en mesure de décider en

TABLE 2 – Résultats expérimentaux (#P, #N, #E pour respectivement le nombre de propagations, de noeuds et d'échecs)

		solveur sans			solveur avec			
instances	lieur	#P	#N	#E	gain	#P	#N	#E
`counter4_2.qbf`	$\exists^{12}\forall^{8}$	5630	556	3	10%	3911	205	3
`counter4_3.qbf`	$\exists^{16}\forall^{12}$	80326	8596	7	100%	54176	3369	7
`counter4_4.qbf`	$\exists^{20}\forall^{16}$	1237107	136431	15	190%	741592	39066	15
`counter4_5.qbf`	$\exists^{24}\forall^{20}$	19857976	2179745	31	160%	13065634	827942	31
`counter4_6.qbf`	$\exists^{28}\forall^{24}$	318342127	34935929	65	190%	203660692	11905482	65
`counter5_2.qbf`	$\exists^{15}\forall^{10}$	17102	2114	3	30%	9844	557	3
`counter5_4.qbf`	$\exists^{24}\forall^{20}$	16623813	2134043	15	320%	8610737	483418	15
`counter6_2.qbf`	$\exists^{18}\forall^{12}$	56410	8296	3	110%	24724	1497	3
`counter6_4.qbf`	$\exists^{30}\forall^{24}$	207892331	31010694	15	∞	91037164	5557142	15
`counter7_2.qbf`	$\exists^{21}\forall^{14}$	198683	32942	3	270%	63115	4017	3
`counter8_2.qbf`	$\exists^{16}\forall^{24}$	719668	131380	3	750%	161471	10829	3
`ring4_2.qbf`	$\exists^{21}\forall^{14}$	6962131	1053640	516	230%	2875645	295639	433
`ring5_2.qbf`	$\exists^{24}\forall^{16}$	26652296	4291339	614	410%	8600128	823310	523
`ring6_2.qbf`	$\exists^{27}\forall^{18}$	101449882	17590062	712	620%	25932045	2434105	621
`semaphore3_2.qbf`	$\exists^{27}\forall^{18}$	54782065	1304353	23	100%	31154191	652233	23
`semaphore_2.qbf`	$\exists^{20}\forall^{14}$	2691738	80409	23	90%	1571193	40257	23

moins de 30 minutes) ; les septième, huitième et neuvième colonnes rapportent respectivement le nombre de propagation, le nombre de nœuds de l'arbre de recherche et le nombre d'échecs pour le solveur *avec* la contrainte d'équivalence. La principale conclusion est la suivante : le solveur avec contrainte d'équivalence est *toujours* plus rapide que le solveur sans celle-ci.

5 Conclusion

Nous avons défini dans cet article une nouvelle contrainte, la contrainte d'équivalence, et présenté comment le problème de validité des QBF pouvait être vu comme un CSP associé à un algorithme de recherche quantifié. Tout élément retiré par la contrainte classique d'égalité l'est egalement par notre contrainte d'équivalence. Mais elle permet aussi de supprimer des éléments supplémentaires non filtrés par la contrainte usuelle. Le filtrage réalisé par la contrainte d'équivalence est donc plus puissant que celui de la contrainte d'égalité. Dans nos résultats expérimentaux, la résolution est plus rapide si la contrainte d'équivalence est présente que si elle est omise.

La puissance du filtrage réside dans sa capacité à éliminer le maximum de valeurs ne pouvant faire partie d'une solution du problème. Dans le cas de domaines booléens, le filtrage peut au plus retirer une valeur par domaine sans détecter d'inconsistance. Dans ce sens, le filtrage est sans doute plus performant pour des domaines de plus grandes tailles. Notre contrainte d'équivalence dépasse le cadre des domaines booléens et peut être étendue à des domaines de plus grande taille. Nous pensons donc à l'adapter pour te tels domaines et espérons que là aussi les performances du solveur s'en trouveraient améliorées.

Références

[1] K.R. Apt. The essence of constraint propagation. *Theoretical Computer Science*, 221(1-2) :179–210, 1999.

[2] 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.

[3] M. Benedetti. Evaluating QBFs via Symbolic Skolemization. In *Proceedings of the 11th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR'05)*, pages 285–300, 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 *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] L. Bordeaux and E. Monfroy. Beyond NP : Arc-consistency for quantified constraints. In *Proceedings of the 8th International Conference on Principles and Practice of Constraint Programming (CP'02)*, pages 371–386, 2002.

[8] M. Cadoli, A. Giovanardi, and M. Schaerf. Experimental Analysis of the Computational Cost of Evaluating Quantified Boolean Formulae. In *Proceedings of the 5th Conference of the Italian Association for Artificial Intelligence (AIIA'97)*, pages 207–218, 1997.

[9] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Communication of the ACM*, 5 :394–397, 1962.

[10] U. Egly, M. Seidl, and S. Woltran. A solver for QBFs in negation normal form. *Constraints*, 14(1) :38–79, 2009.

[11] I.P Gent, P. Nightingale, A. Rowley, and K. Stergiou. Solving quantified constraint satisfaction problems. *Journal of Artificial Intelligence*, 172(6-7) :738–771, 2008.

[12] E. Giunchiglia, M. Maratea, A. Tacchella, and D. Zambonin. Evaluating search heuristics and optimization techniques in propositional satisfiability. In *Proceedings of the 1st International Joint Conference on Automated Reasoning (IJCAR'01)*, pages 347–363, 2001.

[13] E. Giunchiglia, M. Narizzano, and A. Tacchella. Quantified Boolean Formulas satisfiability library (QBFLIB), 2001. `www.qbflib.org`.

[14] 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.

[15] A. Goultiaeva, V. Iverson, and F. Bacchus. A Compact Representation for Syntactic Dependencies in QBFs. In *Proceedings of the 12th International Conference on Theory and Applications of Satisfiability Testing (SAT'09)*, pages 412–426, 2009.

[16] H. Kleine Büning, M. Karpinski, and A. Flögel. Resolution for quantified Boolean formulas. *Information and Computation*, 117(1) :12–18, 1995.

[17] F. Lonsing and A. Biere. Nenofex : Expanding NNF for QBF Solving. In *Proceedings of the Eleventh International Conference on Theory and Applications of Satisfiability Testing (SAT'08)*, pages 196–210, 2008.

[18] G. Pan and M.Y. Vardi. Symbolic Decision Procedures for QBF. In *International Conference on Principles and Practice of Constraint Programming*, 2004.

[19] D.A. Plaisted, A. Biere, and Y. Zhu. A satisfiability procedure for quantified Boolean formulae. *Discrete Applied Mathematics*, 130 :291–328, 2003.

[20] J. Rintanen. Constructing conditional plans by a theorem-prover. *Journal of Artificial Intelligence Research*, 10 :323–352, 1999.

[21] A. Sabharwal, C. Ansótegui, C.P. Gomes, J.W. Hart, and B. Selman. QBF Modeling : Exploiting Player Symmetry for Simplicity and Efficiency. In *Proceedings of the 9th International Conference on Theory and Applications of Satisfiability Testing (SAT'06)*, pages 382–395, 2006.

[22] C. Schulte and G. Tack. Views and Iterators for Generic Constraint Implementations. In *Proceedings of the 11th International Conference on Principles and Practice of Constraint Programming (CP'05)*, pages 817–821, 2005.

[23] 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.

[24] I. Stéphan. Boolean Propagation Based on Literals for Quantified Boolean Formulae. In *Proceedings of the 17th European Conference on Artificial Intelligence (ECAI'06)*, pages 452–456, 2006.

[25] I. Stéphan. Sémantique et calcul syntaxique pour les formules booléennes quantifiées. In *Actes des Quatrièmes Journées Francophones de Programmation par Contraintes (JFPC'09)*, pages 275–284, 2009.

[26] L.J. Stockmeyer. The polynomial-time hierarchy. *Theoretical Computer Science*, 3 :1–22, 1977.

[27] G. S. Tseitin. On the complexity of derivation in propositional calculus. In A. O. Slisenko, editor, *Studies in Constructive Mathematics and Mathematical Logic, Part 2*, pages 115–125. Consultants Bureau, New York, 1970.

[28] L. Zhang. Solving QBF with combined conjunctive and disjunctive normal form. In *National Conference on Artificial Intelligence (AAAI'06)*, 2006.

LocalSolver 1.x *
Un solveur boîte noire à base de recherche locale pour la programmation 0-1

Thierry Benoist[1], **Bertrand Estellon**[2], **Frédéric Gardi**[1], **Romain Megel**[1], **Karim Nouioua**[2]

[1] Bouygues e-lab, Paris

[2] Laboratoire d'Informatique Fondamentale - CNRS UMR 6166,
Faculté des Sciences de Luminy - Université Aix-Marseille II, Marseille

{tbenoist,fgardi,rmegel}@bouygues.com {estellon,nouioua}@lif.univ-mrs.fr

Résumé

Cet article présente LocalSolver 1.x, un solveur boite noire à base de recherche locale pour la programmation 0-1 généralisée. Ce logiciel permet aux praticiens de la recherche opérationnelle de se concentrer sur la modélisation de leur problème dans un formalisme simple, pour ensuite en confier la résolution à un solveur basé sur des techniques efficaces et robustes de recherche locale ("*model & run*"). Après une présentation du formalisme de modélisation et du fonctionnement interne de LocalSolver, nous en démontrons l'efficacité par une vaste étude expérimentale.

Abstract

This paper introduces LocalSolver 1.x, a black-box local-search solver for general 0-1 programming. This software allows OR practitioners to focus on the modeling of the problem using a simple formalism, and then to leave its actual resolution to a solver based on efficient and reliable local-search techniques ("*model & run*").Having outlined the modeling formalism and the main technical features behind LocalSolver, its effectiveness is demonstrated through an extensive computational study.

*Le travail de B. Estellon et K. Nouioua est en partie financé par le projet ANR OPTICOMB (ANR BLAN06-1-138894). T. Benoist, F. Gardi, R. Megel remercient Etienne Gaudin, directeur du e-lab de Bouygues, pour son soutien et ses encouragements.

1 Introduction

En optimisation combinatoire, les techniques de recherche arborescente consistent à explorer l'espace des solutions par une instantiation itérative des variables composant le vecteur solution. Leur efficacité en pratique repose sur leur capacité à élaguer l'arbre de recherche, qui est de taille exponentielle dans le pire des cas. Fondée sur ces techniques, la Programmation Linéaire en Nombres Entiers (PLNE) est sans nul doute un des outils les plus puissants de la Recherche Opérationnelle. Bien que limitée face à des problèmes combinatoires de grande taille, son succès auprès des praticiens est notamment dû à la simplicité d'utilisation des solveurs PLNE : l'ingénieur modélise son problème comme un programme linéaire en nombres entiers et le solveur le résout par *branch & bound* (& *cut*). Ainsi, une tendance récente en Programmation par Contraintes (PPC) est de promouvoir la conception de solveurs PPC autonomes [19]. En effet, cette approche "*model & run*", lorsqu'elle s'avère effective, réduit considérablement les efforts de développement et de maintenance des logiciels d'optimisation.

D'un autre côté, la Recherche Locale (LS, de l'anglais *Local Search*) consiste à appliquer de façon itérative des modifications (appelés mouvements) à une solution de façon à améliorer celle-ci. Bien qu'incomplète, ces techniques sont appréciés des chercheurs opérationnels parce que permettant d'obtenir des solutions de qualité en des temps très courts (de l'ordre de la minute). Cependant, concevoir et implémenter

des algorithmes de recherche locale n'est pas chose facile. La couche algorithmique dédiée à l'évaluation des mouvements est particulièrement difficile à mettre en oeuvre, parce qu'elle requiert à la fois une expertise en algorithmique et une dextérité en programmation informatique. Pour une synthèse sur la recherche locale et ses applications, le lecteur est invité à consulter le livre édité par Aarts et Lenstra [1].

Cet article introduit LocalSolver 1.x, un solveur "boîte noire" pour la programmation 0-1 généralisée (non linéaire). Ce logiciel permet praticien de se concentrer sur la modélisation de son problème, et de laisser sa résolution à un solveur basé sur des algorithmes de recherche locale efficaces et fiables. Débuté en 2007, ce projet vise à offrir une approche de type "*model & run*" pour la résolution des problèmes d'optimisation combinatoire hors de portée des solveurs autonomes de PLNE ou PPC. La version actuelle (LocalSolver 1.1) permet d'attaquer une classe restreinte (mais importante) de problèmes d'optimisation combinatoire : *assignment, partitioning, packing, covering*. Distribuée gratuitement sous licence BSD [1], les binaires du logiciel sotn disponibles pour les architectures x86 et pour les trois systèmes d'exploitations Linux 2.6, Mac OS X 10.5 (Leopard), Windows XP. Le logiciel peut être utilisé pour l'enseignement, la recherche et même à des fins commerciales sans autorisation des auteurs.

Le papier est organisé comme suit. Après un état de l'art sur le sujet, le formalisme LSP associé à LocalSolver 1.x est présenté. Ensuite, le logiciel et les idées maîtresses sur lesquelles il repose sont détaillés. Afin de démontrer l'efficacité de notre solveur, nous présentons dans les grandes lignes les résultats d'une vaste étude expérimentale réalisée sur une dizaine de problèmes académiques et industriels.

2 Travaux connexes et contributions

Une heuristique de recherche locale se conçoit en trois couches [12] : stratégie de recherche et (méta)heuristiques, mouvements, algorithmique d'évaluation. Nos expériences passées dans la conception et l'implémentation d'algorithmes de recherche locale [3, 10, 11, 12] nous ont convaincus que négliger l'une de ces trois couches pouvait entraîner une baisse significative des performances de l'algorithme. Ainsi la conception et l'implémentation d'une recherche locale est une tâche longue et complexe pour les praticiens de la recherche opérationnelle.

La plupart des outils ou des composants réutilisables proposés pour faciliter l'implémentation de recherche locale prennent la forme d'un *framework* pour traiter la couche haute, c'est-à-dire les métaheuristiques (voir par exemple [5, 7]). Ainsi, les mouvements et les algorithmes d'évaluation incrémentale associés restent à la charge de l'utilisateur tandis que le rôle du *framework* est d'appliquer la métaheuristique sélectionnée. Malheureusement, concevoir les mouvements et implémenter les algorithmes d'évaluation représente la plus grande part du travail (et du code source résultant). En effet, nous avons observé que le travail relatif à ces deux couches correspondait respectivement à 30 % et 60 % des temps de développement. Par conséquent, ces *frameworks* ne répondent pas aux besoins majeurs des praticiens.

Deux logiciels visent à répondre à ces besoins : Comet Constraint-Based Local Search (CBLS) [24] (et son ancêtre Localizer [15]) et iOpt [28]. En effet, ces logiciels permettent une évaluation automatique des mouvements, l'implémentation de ces mouvements et de l'heuristique restant à la charge de l'utilisateur. *À notre connaissance, aucun solveur "boîte noire" efficace basé sur la recherche locale n'est disponible à ce jour pour traiter des problèmes réels d'optimisation combinatoire de grandes tailles, comme on en connaît en PLNE ou PPC.* Van Hentenryck et Michel [25] ont publié récemment un article décrivant un synthétiseur d'heuristiques de recherche locale à partir de modèles de haut niveau, mais cette fonctionnalité n'est pas encore disponible dans Comet. Un algorithme tabou générique à base de *swaps* [9, pp. 330–331] est disponible dans Comet CBLS 2.1, et peut être utilisé comme une boîte noire pour aborder des modèles entiers.

Notons également que les meilleurs prouveurs SAT reposent sur des techniques de recherche locale (voir par exemple *Walksat* [23] et WSAT(OIP) [29]), mais ces solveurs ne traite pas la programmation 0-1 généralisée et sont donc rarement utilisés par les praticiens de la recherche opérationnelle.

Notre approche de la recherche locale autonome est guidé par le principe fondamental suivant : *le solveur doit faire ce que le praticien ferait.* Ceci constitue une différence majeure avec les frameworks et solveurs cités plus haut : LocalSolver effectue des mouvements structurés tendant à maintenir la faisabilité des solutions à chaque itération, et dont l'évaluation est accélérée en exploitant les invariants induits par la structure du problème.

Ainsi, *les spécificités majeures de LocalSolver 1.x sont : un formalisme mathématique simple pour modéliser le problème de façon appropriée pour une résolution par recherche locale, et un solveur boîte-noire par recherche locale focalisé sur la faisabilité et l'efficacité des mouvements.*

1. `http://www.localsolver.com`

3 Le formalisme LSP

Le formalisme de modélisation de LocalSolver (appelé LSP pour "Local Search Programming") est proche de formalismes classiques de la programmation mathématique comme la programmation entière 0-1 ou la programmation pseudo booléenne mais est enrichi des opérateurs mathématiques usuels, ce qui le rend facile à prendre en main pour des praticiens de la recherche opérationnelle. Dans le format LSP (*Local Search Programming*), un programme consiste en : des variables de décision, des variables intermédiaires, des contraintes et des objectifs. Bien entendu, le langage de modélisation possède son équivalent en terme d'interface de programmation dans la librairie C++ LocalSolver. Comme exemple, nous donnons le modèle d'un problème jouet de type *bin packing*. Nous avons 3 paquets x, y, z de hauteur 2, 3, 4 respectivement à arranger en 2 piles A et B, sachant que cette dernière contient déjà un paquet de hauteur 5. L'objectif est de minimiser la hauteur de la plus grande pile.

```
xA <- bool(); yA <- bool(); zA <- bool();
xB <- bool(); yB <- bool(); zB <- bool();
constraint booleansum(xA, xB) = 1;
constraint booleansum(yA, yB) = 1;
constraint booleansum(zA, zB) = 1;
hauteurA <- sum(2xA, 3yA, 4zA);
hauteurB <- sum(2xB, 3yB, 4zB, 5);
objectif <- max(hauteurA, hauteurB);
minimize objectif;
```

si le paquet x est placé dans la pile A. Dans cette version 1.0, seuls les variables de décision booléennes sont autorisés, rendant le formalisme proche de la programmation entière 0-1. Ensuite, l'opérateur `<-` est utilisé pour définir des variables intermédiaires (par exemple, la hauteur de chaque pile), qui peuvent être booléennes ou entières. Le mot-clé `constraint` préfixe la déclaration des contraintes ; ici trois contraintes assurent que chaque paquet est affecté à une et une seule pile. De la même façon, le mot-clé `minimize` préfixe la déclaration de l'objectif du programme.

Formellement, la syntaxe BNF d'un programme LSP est :

$$
\begin{array}{rcl}
< lsp > & ::= & (line) \\
< line > & ::= & [< modifier >][< naming >] \\
 & & < expression > \texttt{;} \\
< modifier > & ::= & \texttt{minimize}|\texttt{maximize}| \\
 & & \texttt{constraint} \\
< naming > & ::= & < identifier > \texttt{<-}
\end{array}
$$

Les différents types d'expression sont détaillés dans la section suivante. Notons que l'ordre des lignes dans le programme est libre, excepté lorsque l'on définit plusieurs objectifs.

3.1 Variables décisionnelles et intermédiaires

Toutes les variables de décision doivent être déclarées dans le programme. Cela est fait à l'aide de l'opérateur `bool()`, déclarant une variable booléenne. Les variables booléennes sont traités comme des entiers, avec la convention faux=0 et vrai=1. Nous insistons sur le fait que seuls les variables booléennes sont autorisées comme variables décisionnelles dans cette version de LocalSolver.

Des expressions peuvent être construites à partir de ces variables en utilisant d'opérateurs logiques, arithmétiques et relationnels :

$$
\begin{array}{rcl}
< expression > & ::= & < identifier > \mid < scalar > \mid \\
 & & < scalar >< expression > \mid \\
 & & < operator > \texttt{(}[< arglist >]\texttt{)} \mid \\
 & & < expression >< comparator > \\
 & & < expression > \\
< arglist > & ::= & < expression > [\texttt{,} < arglist >] \\
< operator > & ::= & \texttt{bool} \mid \texttt{and} \mid \texttt{or} \mid \texttt{xor} \mid \texttt{not} \mid \texttt{if} \mid \\
 & & \texttt{sum} \mid \texttt{booleansum} \mid \texttt{min} \mid \texttt{max} \mid \\
 & & \texttt{product} \mid \texttt{divide} \mid \texttt{modulo} \mid \\
 & & \texttt{distance} \mid \texttt{abs} \mid \texttt{square} \\
< comparator > & ::= & \texttt{<} \mid \texttt{<=} \mid \texttt{>} \mid \texttt{>=} \mid \texttt{=} \mid \texttt{!=}
\end{array}
$$

où $< scalar >$ est un entier et $< identifier >$ un nom de variable.

En résumé, LocalSolver utilise une syntaxe fonctionnelle (seuls les opérateurs relationnels ne sont pas préfixés), sans limitation sur l'imbrication des expressions. Des variables intermédiaires peuvent être introduites via l'opérateur `<-`, soit pour accroître la lisibilité du modèle ou pour réutiliser des expressions dans différentes lignes. Certains opérateurs ont des contraintes sur le nombre et le type de leurs opérandes. Par exemple, l'opérateur `not` ne prend qu'un seul argument de type booléen. L'opérateur `if` prend exactement trois arguments, le premier étant nécessairement booléen : `if(condition, value_if_true, value_if_false)`. Les expressions booléennes étant considérées comme des variables 0/1, elles peuvent être utilisées comme opérandes d'opérateurs entiers.

L'introduction des opérateurs logiques, arithmétiques et relationnels a deux avantages importants dans un contexte de recherche locale : l'expressivité et efficacité. Avec ce type d'opérateurs mathématiques de bas-niveau, modéliser est plus facile qu'avec la syntaxe basique de la PLNE, tout en restant rapidement assimilable par les moins expérimentés (en particulier, par les ingénieurs qui ne sont pas à l'aise en programmation informatique). D'un autre côté, les invariants induits par ces opérateurs peuvent être exploités par les algorithmes internes du solveur pour accélérer la recherche locale.

3.2 Contraintes et objectifs

Toute expression booléenne peut être contrainte en préfixant la ligne par `constraint`. Une instantiation des variables est valide si et seulement si toutes les expressions contraintes ont comme valeur 1, c'est-à-dire sont satisfaites. Lors de la modélisation d'un problème l'utilisateur doit garder à l'esprit que la recherche locale n'est pas adaptée pour traiter des problèmes fortement contraints. Si des contraintes métier sont susceptibles de ne pas être satisfaites, alors il est recommandé de les transférer dans la fonction objectif (comme contraintes molles) plutôt que de laisser comme contraintes dures. De plus, LocalSolver offre une fonctionnalité facilitant ce type de modélisation : les objectifs à optimiser dans un l'ordre lexicographique.

Au moins un objectif doit être défini, en utilisant le modificateur `minimize` ou `maximize`. Toute expression peut être utilisée comme objectif. Si plusieurs objectifs sont définis, ils sont interprétés comme une fonction objectif lexicographique. L'ordre lexicographique est induit par l'ordre dans lequel sont déclarés les objectifs. Par exemple, dans le problème d'ordonnancement de véhicules [11], lorsque l'objectif est de minimiser les violations sur les contraintes de ratio et en second lieu le nombre de changements de couleurs dans la séquence, la fonction objectif peut être directement spécifiée comme suit : `minimize ratio_violations; minimize color_changes;`. Ceci permet d'éviter le travers classique de la PLNE où un grand coefficient est utilisé pour simuler l'ordre lexicographique : `minimize 1000 ratio_violations + color_changes;`. Notons que le nombre d'objectifs n'est pas limité et que ceux-ci peuvent avoir différentes directions (minimisation ou maximisation).

4 Recherche locale autonome

La ligne de commande pour résoudre le problème jouet présenté ci-dessus, en laissant 1 seconde de temps de résolution à LocalSolver, est :

```
localsolver.exe io_lsp=toy.lsp hr_timelimit=1
io_solution=toy.sol
```

Ensuite, l'affichage de la console devrait ressembler à celui-ci :

```
Parsing LSP file toy.lsp...
25 nodes, 6 booleans
3 constraints, 1 objectives
1 phases, 2 threads
*** Initial feasible solution : obj = ( 9 )
*** Solve phase 1 over 1
Running phase for 1 sec and 4294967295 itr using
descent
* Thread 0 : [ 2 / 2 / 260000 ] moves in 0 sec,
obj = ( 7 ) in 0 sec and 3 itr
* Thread 1 : [ 3 / 4 / 260000 ] moves in 0 sec,
obj = ( 7 ) in 0 sec and 5 itr
*** Best solution : obj = ( 7 ) in 0 sec and 3
itr
Writing solution in file toy.sol...
```

Par défaut, LocalSolver 1.1 effectue une descente standard [1] en utilisant tous les mouvements autonomes disponibles. Une heuristique de recuit simulé peut également être sélectionnée via les options de la ligne de commande. Le nombre de threads est paramétrable (2 par défaut). Plusieurs résolutions sont alors exécutées en parallèle avec des graines différentes pour le générateur pseudo-aléatoire et leurs résultats sont périodiquement synchronisés. Finalement la meilleure solution est retournée une fois la limite de temps atteinte. Le parallélisme doit moins être vu comme accélérateur de la recherche que comme un moyen d'augmenter la robustesse du solveur. Quant aux mouvements autonomes, ils sont choisis aléatoirement sur la base d'une distribution non uniforme qui peut évoluer au cours de la recherche en fonction de leurs taux d'acceptation et d'amélioration.

Le coût de la solution initiale (admissible) trouvé par LocalSolver est 9. Cette solution est obtenue par un simple algorithme glouton randomisé. Comme expliqué plus haut, LocalSolver n'est pas conçu pour optimiser des problèmes fortement contraints. Par conséquent si cet algorithme d'initialisation ne trouve par de solution faisable, il appartient à l'utilisateur de transformer une de ses contraintes en un objectif de premier niveau. Notons qu'il s'agit là d'une différence fondamentale avec les approches CBLS, dans lesquelles une mesure de violation est définie sur chaque contrainte. Nous considérons que de telles relaxations sont de la responsabilité de l'utilisateur. Typiquement pour des problèmes d'affectation de fréquences, l'ingénieur pourra choisir entre affecter une fréquence à tous les liens tout en minimisant les interférences ou assurer l'absence d'interférence tout en minimisant le nombre de liens sans fréquence affectée.

La meilleure solution, trouvée par recherche locale après 3 itérations (et 0 secondes) par le thread 0, a un coût 7. Durant la seconde de temps alloué, LocalSolver a effectué 260 000 itérations, ce qui correspond au nombre total de mouvements tentés et de fait au nombre de solutions visitées durant la recherche. Parmi ces mouvements, 4 ont été validés sur le thread 1 et 3 ont été (strictement) améliorant. Ces statistiques

sont détaillés par mouvement, ce qui peut être utilisé pour ajuster le paramétrage du solveur. Enfin, LocalSolver crée le fichier "toy.sol" contenant la solution suivante : `xA=0; yA=1; zA=1; xB=1; yB=0; zB=0;`

```
x1 <- bool();
x2 <- bool();
x3 <- bool();
y1 <- bool();
y2 <- bool();
y3 <- bool();
sx <- booleansum(x1, x2, x3);
sy <- booleansum(y1, y2, y3);
constraint sx <= 2;
constraint sy >= 2;
obj <- max(sx, sy);
minimize obj;
```

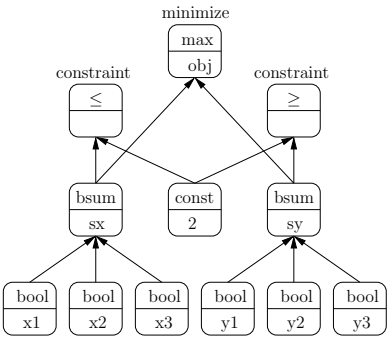


FIGURE 1 – Le graphe acyclique orienté (DAG) induit par un modèle simple. Pour chaque noeud, le type (resp. nom) du noeud est donné au dessus (resp. au dessous). Ici "bsum" signifie `booleansum`.

Un programme LSP, comme défini ci-dessus, peut être représenté par un graphe orienté acyclique (DAG, de l'anglais *Directed Acyclic Graph*), dont les racines sont les variables de décision et les feuilles sont les contraintes et objectifs (voir Fig. 1). Ensuite, les opérateurs utilisés pour modéliser le problème induisent les nœuds internes du DAG. Ces nœuds internes correspondent aux *invariants* ou *one-way constraints* dans des logiciels comme iOpt [28] ou Comet [24]. Selon cette représentation, une solution correspond a une instantiation des variables racines. Ainsi, appliquer des mouvements à la solution courante consiste à modifier les valeurs courantes des variables de décision (racines) et évaluer les valeurs des contraintes et objectifs (feuilles) par propagation des modifications dans le DAG.

L'architecture de LocalSolver est conçue en 3 couches suivant la méthodologie proposée par les auteurs [12] pour l'ingénierie d'heuristiques de recherche locale performantes : (méta)heuristiques, mouvements, évaluation. Résolument orienté vers la simplicité et l'efficacité, la conception et l'implémentation de LocalSolver a requis un effort considérable en terme d'ingénierie logicielle et algorithmique, ne pouvant être entièrement exposé ici. Ainsi, nous concentrerons notre présentation sur deux aspects cruciaux du solveur : l'évaluation des algorithmes et les mouvements autonomes.

4.1 Mouvements autonomes

Comme suggéré en introduction, notre but ultime est d'effectuer automatiquement les mouvements qu'un praticien aurait conçus pour résoudre son problème. Le mouvement le plus simple est le "K-flip" qui inverse aléatoirement les valeurs de K variables de décision (binaires). Cependant la structure du modèle permet souvent au solveur de concevoir des mouvements plus appropriés. Par exemple, lorsqu'une contrainte est posée sur une somme de variables binaires, un mouvement naturel consiste à modifier deux booléens de la somme dans des directions opposées, préservant ainsi la valeur de cette somme. Nous montrerons dans cette section que LocalSolver est également capable d'exploiter des structures plus complexes, en appliquant des mouvements autonomes qui peuvent être vus comme des chaînes d'ejection appliquées à l'hypergraphe induit par les variables booléennes et les contraintes (voir [20] pour plus de détail sur les chaînes d'éjection). Ces chaînes d'éjection sont spécialisées pour préserver la faisabilité des contraintes booléennes et sont un composant clé de l'efficacité de LocalSolver 1.x. Prenons l'exemple du problème d'ordonnancement de véhicules [10, 11] :

Par exemple, considérons le problème d'ordonnancement de véhicules [10, 11] : des voitures doivent être ordonnées sur une ligne de production de façon à minimiser un objectif non linéaire. Ce problème peut être modélisé comme un problème d'affectation (non linéaire) en définissant pour chaque voiture i et position p une variable booléenne $x_{i,p}$. Un voisinage basique pour ce modèle consiste à échanger les positions de deux véhicules. Au niveau du modèle, échanger les positions p et q de deux voitures i et j correspond à flipper successivement les 4 variables booléennes $x_{i,p}$, $x_{i,q}$, $x_{j,q}$, $x_{j,p}$, tout préservant la faisabilité des 4 contraintes de partition où ces variables apparaissent. D'une façon générique, nos mouvements autonomes correspondent à des *k-moves* ou *k-swaps* dans le cadre de problèmes de *packing/covering*.

Appelons une *somme racine* une somme dans laquelle apparaît au moins deux variables de décision binaires (éventuellement multipliées par un scalaire)

et qui est contrainte par un opérateur relationnel. Une structure de données est construite qui liste toutes les sommes racines dans le DAG et pour chaque variable décisionnelle, la liste des sommes racines la contenant. Ensuite, nous maintenons pour chaque somme racine l'ensemble des *booléens croissants*, c'est-à-dire les variables de décision dont la modification augmente la somme, et le complémentaire de cet ensemble (*booléens décroissants*). En utilisant cette structure, nous parvenons à effectuer des mouvements visant à trouver une chaîne alternée de booléens croissants et décroissants tel que deux booléens consécutifs dans cette chaîne appartiennent à la même somme racine. Pour obtenir un cycle alterné, comme décrit dans le précédent paragraphe, la propriété doit être vérifiée circulairement, entre la dernière et la première variable de la chaîne. l'idée maîtresse derrière ces mouvements, appelés *k-Chaînes* ou *k-Cycles*, est de réparer de façon alternée les sommes modifiés, en appliquant une modification opposée à chaque étape. En préservant la satisfaction des contraintes sur les sommes, les *k-Chaînes* et les *k-Cycles* tendent à maintenir la faisabilité de la solution, ce qui est crucial pour l'efficacité de la recherche. Par exemple, lorsque les sommes racines définissent une structure de couplage complet, tout k-Cycle avec k pair sera complété (c'est à dire fermé sans échec) en temps $O(k)$.

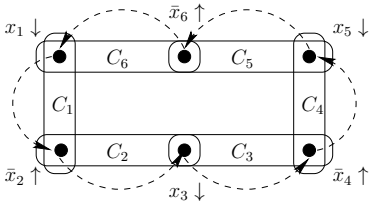


FIGURE 2 – Un 6-Cycle impliquant six variables binaires x_1, x_3, x_5 (dont la valeur actuelle est 1) et $\bar{x}_2$, $\bar{x}_4$, $\bar{x}_6$ (dont la valeur actuelle est 0), et six sommes contraintes $C_1, \ldots, C_6$. Chaque variable appartient à deux sommes (par exemple, x_1 appartient à C_1 et C_6). Diminuons x_1, x_3, x_5 ($\downarrow$) tandis que $\bar{x}_2$, $\bar{x}_4$, $\bar{x}_6$ sont augmentées ($\uparrow$). Ce mouvement préserve les valeurs des sommes et donc la faisabilité des contraintes.

Changer la définition des sommes racines conduit à des variantes qui peuvent être intéressantes en pratique pour accélérer la convergence de la recherche locale. Par exemple, nous pouvons nous concentrer sur des sommes ayant au moins une contrainte parmi leurs successeurs dans le DAG, ou bien sur des sommes dans lesquelles il n'y a que des variables de décision. Pour la sélection de la somme suivante à réparer, on peut également favoriser les sommes intervenant dans une contrainte d'égalité, parce que le mouvement ne peut pas réussir sans réparer ces sommes. Enfin, une autre idée consiste à flipper plus d'une variable par contrainte. Cette extension suit la même logique que la généralisation des chaînes d'éjection aux arbres d'éjection [6]. Cela permet par exemple d'éjecter 2 objets de taille 1 lorsque l'on ajoute un objet de taille 2 dans un ensemble, dans les problèmes de *packing*.

A notre connaissance, la conception de tels mouvements autonomes capables de préserver la faisabilité de la solution est une nouveauté. Ils forment un composant clé de LocalSolver en tant que boîte noire. En effet, par rapport aux voisinages classiques de type k-Flips employés dans les solveurs SAT/PB [23, 29], ils améliorent très largement l'efficacité de la recherche (c'est à dire sa convergence vers des solutions de qualité) sur des problèmes combinatoires structurés de très grande taille (du type de ceux rencontrés en pratique dans les applications de la RO). Notons que les auteurs ont appris récemment que des mouvements autonomes spécifiques sont implémentés dans le logiciel IBM ILOG Transportation PowerOps (TPO) afin de résoudre des problèmes de tournées de véhicules par recherche locale en mode boîte noire [13].

4.2 Algorithmes d'évaluation

Les premiers algorithmes d'évaluation incrémentale ont été introduits dans Localizer [15], l'ancêtre de Comet [24], et iOpt [28]. Ils se basent sur l'exploitation d'invariants sur les opérateurs combinatoires. Bien que ce mécanisme soit connu dans la littérature, nous le présentons ici dans un souci de complétude, et nous illustrons en fin de section les spécificités de notre implémentation.

Chaque nœud du DAG doit implémenter les méthodes suivantes : `init`, `eval`, `commit`, `rollback`. La méthode `init` est responsable de l'initialisation de la valeur du nœud en fonction des valeurs de ses parents, avant la recherche locale. Les structures de données spécifiques attachées au nœud, utilisées pour accélérer son évaluation incrémentale, sont aussi initialisées par cette méthode. Après application d'un mouvement sur les variables de décision, la méthode `eval` est appelée pour réévaluer de façon incrémentale la valeur du nœud, quand celui-ci est impacté durant la propagation. Ensuite, si le mouvement est accepté par l'heuristique, alors la méthode `commit` est appelée sur chaque nœud modifié pour valider les modifications induites par le mouvement. Sinon, le mouvement est rejeté, et la méthode `rollback` est utilisé à la place.

Comme mentionné précédemment, l'évaluation des mouvements est accélérée en exploitant les invariants relatifs à chaque type de nœud [15]. La propagation des modifications est faite par recherche en largeur dans le

DAG, garantissant que chaque nœud est visité au plus une fois. S'appuyant sur un patron de conception du type *observer*, la propagation est réduite aux nœuds impactés : un nœud est dit impacté si l'un de ses parents a été modifié. Par exemple, considérons le nœud $z \leftarrow a < b$ avec une valeur courante égale à vrai. Celui-ci ne sera pas impacté si la valeur de a diminue ou celle de b augmente. Ensuite, à chaque nœud est associé une méthode *eval* appelée pour calculer la nouvelle valeur du nœud impacté. Cette méthode prend en entrée la liste des parents modifiés (c'est-à-dire les nœuds parents dont la valeur a changé). Pour un opérateur linéaire comme `sum`, l'évaluation est simple : si k termes de la somme sont modifiés, alors sa nouvelle valeur est calculée en temps $O(k)$. Mais pour d'autres opérateurs (arithmétiques ou logiques), des accélérations significatives peuvent être obtenues en pratique. Par exemple, considérons le nœud $z \leftarrow \text{or}(a_1, \ldots, a_k)$ avec M la liste des a_i modifiés et T la liste (maintenue) des a_i dont la valeur courante est vrai. Nous pouvons observer que si $|M| \neq |T|$, alors la nouvelle valeur de z est nécessairement vrai. La validité de cette condition permet alors une évaluation en temps $O(1)$, et non plus $O(k)$. En effet, si $|M| < |T|$, alors au moins un parent reste avec une valeur égale à vrai ; sinon, il existe au moins un parent dont la valeur est modifiée de faux à vrai. Notre implémentation est focalisée sur la complexité pratique et expérimentale et non pas seulement sur la complexité au pire cas. Les facteurs constants ont une grande importance : de bons algorithmes et une optimisation du code améliorent significativement la vitesse d'évaluation. Par exemple la propriété mentionnée ci dessus pour maintenir l'opérateur `or` n'est, à notre connaissance, pas employée dans les systèmes Comet ou iOpt. De la même manière, dans Localizer [15, p. 67] maintient l'opérateur `min` en temps $O(\log k)$ avec k le nombre d'opérandes, en utilisant classiquement un tas binaire. Dans LocalSolver nous distinguons deux cas. Si la valeur minimum parmi les opérandes modifiés est inférieure ou égale à la valeur actuelle de l'opérateur `min` ou bien si un support demeure inchangé, alors l'évaluation est faite optimalement en temps $O(|M|)$ avec $|M|$ le nombre de valeurs modifiées. Sinon l'évaluation est faite en temps $O(k)$ time. En pratique le premier cas est de loin le plus fréquent et et nombre d'opérandes modifiés ne dépende pas de la taille du problème ($|M| = O(1)$), ce qui assure une complexité amortie en temps constant pour l'évaluation.

5 Résultats expérimentaux

LocalSolver a été testé sur un benchmark mixant problèmes académiques et industriels, sélectionnés avant de débuter le projet. Nous insistons sur le fait que notre but n'est pas d'atteindre (encore moins dépasser) les meilleurs résultats de la littérature pour tous ces problèmes. *Le but de LocalSolver est d'obtenir en mode boîte noire de bonnes solutions en des temps d'exécution courts (comme le ferait des heuristiques standards de recherche locale), en particulier quand les solveurs arborescents sont incapables d'en trouver une.* Le but de ces expérimentations est de comparer LocalSolver aux solveurs boîte noire existants : IBM ILOG CPLEX 12.2 (le solveur de référence pour la programmation entière) et Comet CBLS 2.1 [9, pp. 330-331] qui, bien que n'étant pas conçu a priori pour un usage boîte noire, offre une recherche tabu générique à base de swaps. IBM ILOG CP Optimizer a été testé également mais n'a pas conduit à des résultats compétitifs sur ces problèmes. Les résultats de solveurs SAT ou PB, inappropriés au traitement de ce type de problèmes structurés d'optimisation, sont également omis.

Pour chacun de problèmes de ce benchmark, les solveurs sont comparés sur la même machine avec le même modèle standard. Toutes les expérimentations ont été conduites sur un ordinateur standard équipé du système Windows XP 32 bits et du processeur Intel Core 2 Duo T7600 (2.33 GHz, RAM 2 Go, L2 4 Mo, L1 64 ko). Notons que deux coeurs seulement sont disponible sur cet ordinateur. Quant au modèle il est simplement adapté à la grammaire de chaque solver. Ainsi les modèles LSP et PLNE sont exprimés à l'aide de variables de décision binaires alors que les modèles CP et CBLS utilisent des variables entières et les contraintes globales disponibles. De même un opérateur *max* par exemple est natif en CBLS et LocalSolver mais sera exprimé comme un ensemble d'inégalités dans le modèle PLNE équivalent. Il est important de souligner qu'aucun élement spécifique n'a été ajouté au delà de ces nécessaires transformations (par d'inégalités valides par exemple).

L'efficacité d'un solveur autonome est une combinaison de plusieurs facteurs comme son implémentation, ses algorithmes internes, sa stratégie de recherche, ses mouvements ou coupes par défaut, sa recherche d'une solution initiale, etc. Ici chaque solveur est lancé avec ses paramètres par défaut, sauf mention contraire. En particulier aucune aucune solution initiale n'est fournie à LocalSolver ou Comet et aucune stratégie de recherche ou mouvements ne sont spécifiés : ces choix sont la responsabilité des solveurs boîte noire.

Chaque problème traité dans ce benchmark est brièvement décrit et le modèle choisi est cité ou esquissé. Les résultats obtenus par les différents solveurs sont donnés sur un ensemble représentatif d'instances, ainsi que (en tant que référence) les meilleurs résultats connus dans la littérature, généralement obtenus par

des heuristiques de recherche locale. Tous les résultats présentés ici ont été rigoureusement validés, en particulier le respect des contraintes et l'exactitude de l'objectif ont été vérifiés en dehors du modèle mathématique. Tout le matériel utilisé pour ce benchmark (code, modèles, résultats) est disponible sur demande.

Dans tous les tableaux ci-dessous, la ligne "LocalSolver 1.1" correspond aux résultats obtenus par LocalSolver 1.1 ; la ligne "CPLEX 12.2" correspond aux résultats obtenus par IBM ILOG CPLEX 12.2 et la ligne "CBLS Comet 2.1" correspond aux résultats obtenus par la recherche tabu générique de Comet 2.1. L'origine de la ligne "state of the art" sera donnée pour chaque problème.

5.1 Ordonnancement de véhicules

Le problème d'ordonnancement de véhicules [14] consiste à ordonner des véhicules sur une chaîne d'assemblage tout en minimisant les violations sur des contraintes de ratio. Pour LocalSolver et CPLEX, l'affectation des véhicules aux positions est modélisée par des variables booléennes et les pénalités sur chaque ratio sont additionnées (see [10]). En Comet nous utilisons des variables de décision entières et non booléennes. En outre une contrainte "sequence" est disponible dans ce language pour modéliser exactement les pénalités d'ordonnancement de véhicules. Cette contrainte globale a été nécessaire pour obtenir les résultats présentés ci-dessous (sans cette contrainte les résultats de Comet sont deux fois plus grands sur les plus grandes instances de ce problème de minimization).

Un échantillon de résultats est présenté pour 5 instances sur la Table 1 en fin de papier : 10-93 (100 véhicules, 5 options, 25 classes), 200-01 (200 véhicules, 5 options, 25 classes), 300-01 (300 véhicules, 5 options, 25 classes), 400-01 (400 véhicules, 5 options, 25 classes), 500-08 (500 véhicules, 8 options, 20 classes). Les 4 premières instances sont disponibles dans la CSPLib [14] ; la cinquième provient d'un benchmark généré par Perron *et al.* [17]. Dans la table, la ligne "state-of-the-art" correspond aux résultats obtenus avec l'algorithme de recherche locale des auteurs [10], qui obtient les meilleurs résultats sur toutes les instances. Les résultats présentés dans la table du haut (resp. bas) ont été obtenus avec un temps d'exécution limité à 60 (resp. 600) secondes. Le coût de la meilleure solution trouvée est donné et le symbole "x" est écrit si aucune solution n'a été obtenue dans le temps imparti. En résumé, nous observons que les résultats de LocalSolver dépassent ceux des solveurs de PLNE et CBLS, d'autant plus que l'échelle des instances grandit (notons que les instances à 400 et 500 véhicules induisent chacun 10 000 variables de décision booléennes). Nous ne détaillons pas les résultats des solveurs de PPC [16, 17], qui ne s'avèrent pas compétitifs pour traiter ce problème : Peron *et al.* [16, 17] obtiennent par *Large Neighborhood Search* un nombre de violations supérieur à 500 pour l'instance 500-08.

Une version industrielle du problème, intégrant les contraintes et objectifs de l'atelier de peinture, a été proposé par l'entreprise Renault comme sujet du Challenge ROADEF 2005 [11] (compétition internationale organisée tous les deux ans par la Société Française de Recherche Opérationnelle). Dans cette version, trois objectifs lexicographiques sont optimisés : EP = violations sur les contraintes de ratio prioritaires, ENP = violations sur les contraintes de ratio non prioritaires, RAF = le nombre de changements de couleur dans la séquence. La Table 2 contient un échantillon des résultats sur 3 instances : X2 = 023-EP-RAF-ENP-S49-J2 (1260 véhicules, 12 options, 13 couleurs), X3 = 024-EP-RAF-ENP-S49-J2 (1319 véhicules, 18 options, 15 couleurs), X4 = 025-EP-ENP-RAF-S49-J1 (996 véhicules, 20 options, 20 couleurs). Aucun solveur de PLNE/PPC/SAT n'est parvenu à traiter ces instances à ce jour [11]. Par exemple CPLEX 11.2 ne parvient pas à trouver de solution après plusieurs heures de calcul, de même pour Comet. La ligne 'Comet CBLS 2.1 (relaxed)" donne les résultats obtenus avec des modèles dans lesquels les contraintes de "paint limit" sont omises. Ici la recherche locale de l'état de l'art correspond à celle locale qui a remporté le challenge [11] ; notons que la conception et l'implémentation de cet algorithme a demandé près de 150 jours de travail à leurs auteurs. Pour l'instance X2, le modèle LSP contient 516 936 variables dont 374 596 sont des variables de décision binaires (450 Mo de RAM sont alloués durant l'exécution).

Localsolver réalise entre 1.5 et 4.5 million de mouvements par minute, avec un taux d'acceptation entre 5 et 20 % et presque un millier de solutions améliorantes. Observons que les résultats de LocalSolver utilisé sont comparables à ceux de la recherche à voisinages variables de Prandtstetter et Raidl [18] qui mixe mouvements classiques et voisinages larges explorés par PLNE.

5.2 Social golfer

Le problème *social golfer* [14] consiste à affecter des personnes à des groupes sur plusieurs semaines de façon à maximiser le nombre de rencontres uniques. Une fois modélisée la structure de partitionnement pour chaque semaine, la détection des rencontres entre golfeurs est très simple dans les différents solveurs considérés. Ce problème est rencontré chez Bouygues SA lors de la planification des séminaires des managers du Groupe Bouygues. Des échantillons de résultats

sont présentés sur la Table 3. Quatre instances classiques sont traitées (2 faciles et 2 difficiles) : 9-3-11 (9 groupes de 3 golfeurs sur 11 semaines), 10-10-3, 10-9-4, 10-3-13. Dans ce cas, l'objectif est de minimiser le nombre de rencontres en doublon. La cinquième instance, appelée "séminaire", est une instance réelle (120 personnes sur 3 semaines avec des tailles de groupe entre 7 et 9) avec comme des contraintes additionnelles portant sur les groupes et trois objectifs lexicographiques : équilibrer les caractères dans chaque groupe (un caractère peut être la filiale dans laquelle évolue la personne, son sexe, ou encore ses centres d'intérêt), éviter les rencontres indésirables, et maximiser le nombre de rencontres uniques (souhaitées). Pour les instances classiques, la recherche locale de l'état de l'art correspond à l'heuristique tabou implémentée en langage C dans [8], qui détient les records sur presque toutes les instances. Des approches par PPC dédiées et complexes (visant à casser les symétries) permettent d'obtenir des résultats similaires [8]. Pour l'instance réelle, l'état de l'art correspond à l'algorithme de recherche locale implémenté par un des auteurs comme solution opérationnelle : une heuristique de descente en *first-improvement* effectuant des échanges aléatoires à l'évaluation efficace (plus d'un million par seconde). Observons que du fait de la forme quadratique de la fonction objectif (le décompte des rencontres ne peut se faire qu'en utilisant un "et" logique), un tel problème est très difficile à traiter par des techniques de PLNE ; en effet, sa linéarisation induit un très grand nombre de variables (centaines de milliers). En résumé, nous observons qu'en dépit d'un temps d'exécution 10 fois plus élevé, les solveurs de PLNE ne parviennent pas à produire de bonnes solutions. Comme précédemment, LocalSolver ne parvient pas à atteindre les records pour les instances difficiles, mais en est proche.

5.3 Découpe de plaques d'acier

Le problème de découpe de plaques d'acier (connu sous l'appellation anglaise *Steel Mill Slab Design*) [14] est un problème de type *bin packing / cutting stock*, où des commandes d'acier de différentes tailles doivent être découpées dans des plaques de différents capacités tout en minimisant les chutes. En outre chaque commande a une "couleur" et le nombre de couleurs différentes dans une plaque est limité. Le modèle est basé sur l'affectation des commandes au plaques et la taille de chaque plaque est determinée par son contenu (même modèle que [22]). L'instance classique de la CSPLib avec 111 plaques [14] est résolue à l'optimum (coût égal à 0) en moins d'une seconde par LocalSolver, dépassant ainsi les approches proposées jusqu'à présent (PLNE, PPC, SAT) [26]. Notons que le LSP traité par LocalSolver dans ce cas contient 40 739 variables dont 12 321 booléens ; LocalSolver visite près de 100 000 solutions par seconde. Un échantillon des résultats obtenus sur les nouvelles instances proposées par Schaus [2] est donné Table 4. Les 200 instances numérotées de 11-0 à 20-19 ne sont pas mentionnées, car toutes sont résolues à l'optimum en moins d'une seconde. Les meilleurs résultats connus sont obtenus par Heinz [2], en utilisant judicieusement une décomposition de Dantzig-Wolfe qui peut être directement résolue par un solveur de PLNE grâce au nombre raisonnable de colonnes (fortement filtrées grâce aux contraintes de couleur et qui peuvent donc être toutes générées d'emblée). Notons que cette approche est dédiée au instances de la CSPLib : des limites moins basses sur le nombre de couleurs rendraient le nombre de colonnes bien plus grand, imposant alors une exploration de type branch-and-price. LocalSolver parvient ici à rester relativement proches des solutions optimales alors que CPLEX et CBLS en sont très loin. De même, les meilleures approches par PLNE ne sont pas competitives [22].

5.4 Prise de vues par le satellite Spot 5

Le problème de prise de vues par le satellite Spot 5 [27] consiste à sélectionner un sous-ensemble de photos à prendre par le satellite Spot 5 ; le but est de maximiser une fonction de profit sujet à des contraintes de sac-à-dos et d'exclusion mutuelle. Ce programme linéaire en variables binaires s'écrit de la même manière pour CPLEX et pour LocalSolver. Les plus grandes instances abordées dans la littérature (cas multi-orbites) contiennent au plus un millers de photos en entrée, ce qui les rend aujourd'hui efficacement abordables par des solveurs de PLNE. Un échantillon de résultats est présenté Table 5. L'état de l'art correspond ici à l'heuristique tabou de Vasquez et Hao [27] ; de plus, ces auteurs ont montré que leurs résultats se situés à moins d'1 % de l'optimum. La conclusion principale de cette expérience est que LocalSolver reste compétitif face aux solveurs de PLNE lorsque l'échelle des instances, plus petite, devient favorable aux techniques de recherche arborescente.

5.5 Minimum de matériel de coffrage

Le problème de minimisation du matériel de construction (en anglais *minimum formwork stock problem* [4], rencontré chez Bouygues Construction, consiste à minimiser le matériel de coffrage utilisé par un chantier. Une fois décomposé le problème peut être vu comme un problème de couverture dont l'échelle permet une résolution efficace par un solveur de PLNE.

2. `http://becool.info.ucl.ac.be/steelmillslab`

Quelques résultats sont présenté en Table 6. Comme dans le problème précédent, on observe que LocalSolver reste compétitif face à CPLEX.

5.6 Eternity II

Le problème Eternity II [2, 21] est un puzzle aussi ludique que difficile édité par la société TOMY en 2007. Le puzzle consiste à remplir une grille de taille 16×16 avec 256 pièces carrées, dont les quatre côtés sont colorés. Le but est de trouver un placement des pièces sur le plateau de façon à ce que les côté adjacents de toute paire de pièces voisines soient de la même couleur. Un tel problème peut être modélisé comme un problème d'optimisation : affecter toutes les pièces sur le plateau tout en minimisant le nombre de paires violant ces contraintes d'adjacences de couleur. A notre connaissance la meilleure solution connue a 13 violations (sur 480). [21] obtiennent une solution à 22 violations à l'aide d'une recherche tabou à grand voisinage. Notons qu'ils obtiennent des solutions avec presque 70 violations en effectuant uniquement des échanges de pièces au sein d'un recherche tabou. Les variables de décision pour chaque pièce sont sa rotation et sa position sur le plateau (exprimées par des variables booléennes en CPLEX et LocalSolver) et les arêtes incompatibles sont simplement détectées à l'aide des opérateurs disponibles dans chaque solver. Nous obtenons une solution avec 70 violations en une journée de calcul en utilisant LocalSolver 1.1. Dans ce cas, LocalSolver réalise presque un million de mouvements par minute alors que le modèle LSP contient 262 144 variables de décision binaires. Notons que CPLEX ne fournit aucune solution après un jour de calcul alors que la recherche générique de Comet obtient une solution avec 107 violations. Les approches purement PPC ne parviennent pas à descendre sous la barre des 80 violations [21].

6 Conclusion

Les résultats ci-dessus démontre qu'une "programmation par recherche locale" est possible : une paradigme du type "*model & run*" pour la recherche locale peut être obtenu en combinant un langage de modélisation simple et un solveur incrémental efficace basé sur des mouvements autonomes appropriés. Ainsi, la prochaine version de LocalSolver est envisagée suivant plusieurs directions de recherche. Tout d'abord, le formalisme LSP est loin d'être complet : notre préoccupation principale est d'y ajouter la notion d'ensembles sans perdre en simplicité et généricité. Cette étape est cruciale pour attaquer des problèmes complexes d'ordonnancement et de routage. Ensuite, le concept de mouvements autonomes maintenant la faisabilité, qui est une des clés de notre approche, doit être renforcé et développé. Enfin, nous avons planifié d'ajouter plus de métaheuristiques [1] dans la couche supérieure du solveur, en sus de la descente standard et du recuit simulé.

Références

[1] E. Aarts and J.K. Lenstra. Local search in combinatorial optimization. In E. Aarts and J.K. Lenstra, editors, *Local Search in Combinatorial Optimization*, Wiley-Interscience Series in Discrete Mathematics and Optimization. John Wiley & Sons, Chichester, England, UK, 1997.

[2] T. Benoist and E. Bourreau. Fast global filtering for Eternity II. *Constraint Programming Letters*, 3 :35–50, 2008.

[3] T. Benoist, B. Estellon, F. Gardi, and A. Jeanjean. High-performance local search for solving inventory routing problems. In T. Stützle, M. Birattari, and H. Hoos, editors, *Proceedings of SLS 2009, the 2nd International Workshop on Engineering Stochastic Local Search Algorithms*, volume 5752 of *Lecture Notes in Computer Science*, pages 105–109. Springer, 2009.

[4] T. Benoist, A. Jeanjean, and P. Molin. Minimum formwork stock problem on residential buildings construction sites. *4OR-Q J Oper Res*, 7 :275–288, 2009.

[5] S. Cahon, N. Melab, and E.-G. Talbi. ParadisEO : a framework for the reusable design of parallel and distributed metaheuristics. *Journal of Heuristics*, 10(3) :357–380, 2004.

[6] Y. Caseau, F. Laburthe, and G. Silverstein. A meta-heuristic factory for vehicle routing problems. In *Proceedings of CP 1999*, volume 1713 of *Lecture Notes in Computer Science*, pages 144–158. Springer, 1999.

[7] L. Di Gaspero and A. Schaerf. EasyLocal++ : an object-oriented framework for flexible design of local search algorithms. *Software - Practice & Experience*, 33(8) :733–765, 2003.

[8] I. Dotú and P. Van Hentenryck. Scheduling social golfers locally. In *Proceedings of CPAIOR 2005*, volume 3524 of *Lecture Notes in Computer Science*, pages 155–167. Springer, 1999.

[9] Dynadec Decision Technologies Inc., Providence, RI. *Comet 2.1 Tutorial*, March 2010. 581 pages, `http://www.dynadec.com`.

[10] B. Estellon, F. Gardi, and K. Nouioua. Large neighborhood improvements for solving car sequencing problems. *RAIRO Operations Research*, 40(4) :355–379, 2006.

[11] B. Estellon, F. Gardi, and K. Nouioua. Two local search approaches for solving real-life car sequencing problems. *European Journal of Operational Research*, 191(3) :928–944, 2008.

[12] B. Estellon, F. Gardi, and K. Nouioua. High-performance local search for task scheduling with human resource allocation. In *Proceedings of SLS 2009*, volume 5752 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 2009.

[13] D. Fernandez Pons, 2010. Personnal communication.

[14] B. Hnich, I. Miguel, I.P. Gent, and T. Walsh. CSPLib : a problem library for constraints, 2009. `http://www.csplib.org`.

[15] L. Michel and P. Van Hentenryck. Localizer. *Constraints*, 5(1/2) :43–84, 2000.

[16] L. Perron and P. Shaw. Combining forces to solve the car sequencing problem. In *Proceedings of CPAIOR 2004*, volume 3011 of *Lecture Notes in Computer Science*, pages 225–239. Springer, 2004.

[17] L. Perron, P. Shaw, and V. Furnon. Propagation guided large neighborhood search. In *Proceedings of CP 2004*, volume 3258 of *Lecture Notes in Computer Science*, pages 468–481. Springer, 2004.

[18] M. Prandtstetter and G.R. Raidl. An integer linear programming approach and a hybrid variable neighborhood search for the car sequencing problem. *European Journal of Operational Research*, 191(3) :1004–1022, 2008.

[19] J.F. Puget. Constraint programming next challenge : Simplicity of use. In *Proceedings of CP 2004*, volume 3258 of *Lecture Notes in Computer Science*, pages 5–8, 2004.

[20] C. Rego and F. Glover. Local search and metaheuristics. In G. Gutin and A. Punnen, editors, *The Traveling Salesman Problem and Its Variations*, pages 105–109. Kluwer Academic Publishers, Dordrecht, Netherlands, 2002.

[21] P. Schaus and Y. Deville. Hybridation de la programmation par contraintes et d'un voisinage à très grande taille pour Eternity II. In *Proceedings of JFPC 2008*, pages 115–122, 2008.

[22] P. Schaus, P. Van Hentenryck, J.-N. Monette, C. Coffrin, L. Michel, and Y. Deville. Solving steel mill slab problems with constraint-based techniques : CP, LNS, CBLS. to appear in Constraints, 2011.

[23] B. Selman, H. Kautz, and B. Cohen. Local search strategies for satisfiability testing. In D.S. Johnson and M.A. Trick, editors, *Cliques, Coloring, and Satisfiability : 2nd DIMACS Implementation Challenge*, volume 26 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*. AMS, Providence, RI, 1996.

[24] P. Van Hentenryck and L. Michel. *Constraint-Based Local Search*. The MIT Press, Boston, MA, 2005.

[25] P. Van Hentenryck and L. Michel. Synthesis of constraint-based local search algorithms from high-level models. In *Proceedings of AAAI 2007*, pages 273–279, 2007.

[26] P. Van Hentenryck and L. Michel. The steel mill slab design problem revisited. In *Proceedings of CPAIOR 2008*, volume 5015 of *Lecture Notes in Computer Science*, pages 377–381, 2008.

[27] M. Vasquez and J.-K. Hao. A "logic-constrained" knapsack formulation and a tabu algorithm for the daily photograph scheduling of an earth observation satellite. *Computational Optimization and Applications*, 20(2) :137–157, 2001.

[28] C. Voudouris, R. Dorne, D. Lesaint, and A. Liret. iOpt : a software toolkit for heuristic search methods. In *Proceedings of CP 2001*, volume 2239 of *Lecture Notes in Computer Science*, pages 716–730, 2001.

[29] J. Walser, R. Iyer, and N. Venkatasubramanyan. An integer local search method with application to capacitated production planning. In *Proceedings of AAAI 1998*, pages 373–379, 1998.

TABLE 1 – Quelques résultats pour le problème d'ordonnancement de véhicule (minimisation).

time limit : 60 s	10-93	200-01	300-01	400-01	500-08
state-of-the-art	3	0	0	1	0
LocalSolver 1.1	8	8	8	13	18
CPLEX 12.2	6	11	27	17	x
Comet CBLS 2.1	7	8	16	18	91
time limit : 600 s	10-93	200-01	300-01	400-01	500-08
state-of-the-art	3	0	0	1	0
LocalSolver 1.1	6	5	4	6	6
CPLEX 11.2	3	3	11	16	104
Comet CBLS 2.1	7	6	10	18	47

TABLE 2 – Quelques résultats pour le problème d'ordonnancement des véhicules Renault (minimisation). le classement de chaque résultat relativement au 18 finalistes du challenge Roadef est donné entre parenthèses.

time limit : 600 s	X2			
state-of-the-art	0,	192,	66	(1/19)
LocalSolver 1.1	0,	268,	212	(16/19)
Comet CBLS 2.1 (relaxed)	799,	1069,	481	(19/19)

time limit : 600 s	X3			
state-of-the-art	0,	337,	6	(1/19)
LocalSolver 1.1	36,	544,	187	(16/19)
Comet CBLS 2.1 (relaxed)	1447,	1100,	309	(19/19)

time limit : 600 s	X4			
state-of-the-art	0,	160,	407	(1/19)
LocalSolver 1.1	2,	353,	692	(18/19)
Comet CBLS 2.1 (relaxed)	1055,	1888,	651	(19/19)

TABLE 3 – Quelques résultats pour le problème du social golfer (minimisation).

time limit : 60 s	9-3-11	10-10-3	10-9-4	10-3-13	seminar
state-of-the-art	0	0	0	0	1, 0, 1082 = 11 082
LocalSolver 1.1	0	0	4	1	1, 0, 1082 = 11 082
Comet CBLS 2.1	2	0	8	6	x
time limit : 600 s	9-3-11	10-10-3	10-9-4	10-3-13	seminar
CPLEX 12.2	94	140	218	125	3 629 775
Comet CBLS 2.1	1	0	5	3	x

TABLE 4 – Quelques resultats pour le *Steel Mill Slab Design Problem* (minimisation).

time limit : 60 s	2-0	3-0	4-0	5-0	6-0
state-of-the-art	22	5	32	0	0
LocalSolver 1.1	31	5	34	4	8
CPLEX 12.2	178	511	x	x	x
Comet CBLS 2.1	136	135	69	65	42
time limit : 300 s	2-0	3-0	4-0	5-0	6-0
state-of-the-art	28	6	34	0	0
LocalSolver 1.1	40	34	35	3	7
CPLEX 12.2	94	65	x	63	x
Comet CBLS 2.1	124	110	43	58	33

time limit : 60 s	7-0	8-0	9-0	10-0
state-of-the-art	0	0	0	0
LocalSolver 1.1	2	0	0	0
CPLEX 12.2	275	226	229	201
Comet CBLS 2.1	30	26	21	20
time limit : 300 s	7-0	8-0	9-0	10-0
state-of-the-art	0	0	0	0
LocalSolver 1.1	1	0	0	0
CPLEX 12.2	189	226	97	64
Comet CBLS 2.1	33	17	17	15

TABLE 5 – Quelques résultats pour le problème Spot 5 de planification de prises de vues (maximisation).

time limit : 60 s	54	414	509	1401	1403
state-of-the-art	70	22 120	19 125	176 056	176 140
LocalSolver 1.1	69	22 115	19 118	167 064	169 127
CPLEX 12.2	70	22 119	19 125	176 056	176 138
time limit : 60 s	1405	1407	1502	1504	1506
state-of-the-art	176 179	176 245	61 158	124 243	168 247
LocalSolver 1.1	160 177	162 253	61 158	123 240	152 247
CPLEX 12.2	174 181	176 237	61 158	124 243	168 246

TABLE 6 – Quelques résutlats sur le *Minimum Formwork Stock* (minimisation).

time limit : 60 s	site1	site8b	site12b	site13b
LocalSolver 1.1	5 640 326	5 640 398	9 223 040	7 729 336
CPLEX 12.2	5 409 158	5 409 240	8 392 196	7 408 436

Heuristique de choix de variable dirigée par les conflits pour la méthode hybride FC-CBJ et mémorisation de conflits pour un CSP Dynamique de T CSPs

F. Bessaïh **B.Cabon** **Ph.Michelon**

EADS ASTRIUM
, 31 rue des Cosmonautes, 31402 Toulouse, France.
Laboratoire Informatique d'Avignon
, (EA 931), F-84911 Avignon, France.
{fawzi.bessaih,philippe.michelon}@univ-avignon.fr,bertrand.cabon@astrium.eads.net

Résumé

Dans cet article, nous proposons une nouvelle heuristique de choix de variable dynamique, $dom/wdeg_{cbj}$, pour une méthode hybride alliant un schéma prospectif et un schéma rétrospectif, mis en œuvre respectivement par le *Forward-Checking* (FC) et le *Conflict Directed Backjumping* (CBJ). C'est une heuristique dirigée par les conflits et inspirée de l'heuristique *dom/wdeg* proposée par Boussemart[11] pour des méthodes prospectives. Cette nouvelle heuristique permet de réduire le phénomène appelé thrashing en choisissant en priorité les variables conduisant le plus souvent à des situations de conflit. Une telle information est enregistrée en associant un poids à chaque variable. Lors d'une situation de conflit nécessitant un retour arrière, ce poids est incrémenté pour la variable courante et la variable responsable du conflit (identifiée par CBJ). Nous proposons également, dans le cadre d'un CSP dynamique de T CSPs, un mécanisme d'apprentissage par mémorisation des conflits, permettant l'exploitation de l'effort de résolution des CSPs statiques successifs.

Des expérimentations menées dans un cadre industriel et sur une problématique réelle montrent l'efficacité de la nouvelle heuristique et du mécanisme d'apprentissage.

Abstract

In this paper, we propose a new dynamic variable ordering heuristics, $dom/wdeg_{cbj}$, for a hybrid method which combining a look-ahead and a look-back schema, implemented respectively by *Forward-Checking* (FC) and *Conflict Directed Backjumping* (CBJ). This heuristics is a conflict-directed variable ordering heuristics and inspired by the heuristics *dom/wdeg* proposed by Boussemart[11] for a look-ahead methods. This new heuristics allows to reduce thrashing phenomenon by first instantiating variables that have frequently participated in conflict situations. Such information is recorded by associating a weight to each variable. During a conflict situation that requiring a backtrack return, the weight of the current variable and the variable which is responsible for the conflict situation (identified by CBJ) are increased. We also propose, within a context of a Dynamic CSP of T CSPs, a learning mechanism by conflict recording, allowing to exploit the resolution efforts of the successive static CSPs.

Experiments led in an industrial context and on a real problem show the efficiency of the new heuristics and the learning mechanism.

1 Introduction

Depuis l'avènement des algorithmes de recherche arborescente pour la résolution des Problèmes de Satisfaction de Contraintes (CSPs), ces derniers n'ont eu de cesse d'évoluer au travers de nombreuses améliorations. Ces améliorations permettent de limiter l'explosion combinatoire inhérente à ce type d'approches. Elles portent principalement sur des mécanismes mis en œuvre durant la phase de progression « look-ahead »[9], tel que l'heuristique de choix de variable (de valeur) et le mécanisme de filtrage, et durant la phase de régression « look-back »[9] tel que l'analyse

des conflits.

Durant les années 90, l'algorithme prospectif Forward-Checking (FC) [16] associé à la méthode rétrospective Conflict Directed Backjumping (CBJ) [20, 22, 14], et la méthode prospective Maintien de l'Arc Coherence (MAC) [13, 21], étaient considérés comme étant les algorithmes les plus efficaces pour résoudre les instances CSPs. Cependant, la méthode MAC avait un avantage significatif lors de la résolution d'instances difficiles de grande taille [4, 7]. Cet avantage a été remis en question par [2, 15, 19], qui proposent une amélioration du mécanisme rétrospectif en associant une explication non pas à chaque variable mais à chaque valeur, donnant ainsi une capacité de retour arrière plus importante. De telles méthodes rétrospectives semblent surclasser la méthode prospective MAC pour des instances difficiles de grande taille.

Néanmoins, les techniques prospectives ont repris un certain avantage depuis l'introduction, par Boussemart, d'une nouvelle heuristique de choix de variable dirigée par les conflits [11]. Elle permet de traiter plus efficacement les problèmes structurés et non structurés en orientant la recherche vers les parties difficiles ou inconsistantes du problème. L'idée est d'associer un poids à chaque contrainte et d'incrémenter ce poids à chaque fois que cette dernière est violée pendant la recherche. Ces poids permettent ensuite de définir un degré pondéré *wdeg* pour chaque variable. Combiné à la taille du domaine courant *dom*, le degré pondéré *wdeg* permet de définir l'heuristique de choix de variable *dom/wdeg*, qui s'apparente à de la mémorisation de *nogoods*, une forme d'apprentissage à partir des conflits. L'heuristique *dom/wdeg* réussit à combiner avantageusement l'aspect prospectif et rétrospectif des algorithmes de recherche arborescente. Elle permet de limiter le phénomène de trashing (redécouverte des mêmes inconsistances) et est une alternative simple aux techniques de retours arrières intelligents[6].

À notre connaissance, l'étude de l'heuristique *dom/wdeg* n'a été menée que dans le cadre de méthodes prospectives liées à un schéma de recherche de type retour arrière chronologique (backtrack), telles que *FC* et *MAC*. Dans cet article nous proposons une nouvelle heuristique dirigée par les conflits $dom/wdeg_{cbj}$. $dom/wdeg_{cbj}$ est inspirée de *dom/wdeg* et s'applique à une méthode hybride alliant un schéma prospectif et un schéma rétrospectif, mis en œuvre respectivement par le Forward-Checking (FC) et le Conflict Directed Backjumping (CBJ) : *FC-CBJ*. Elle exploite la capacité de *CBJ* à identifier la variable source de l'échec. Pour un coût de mise en œuvre limité, cette nouvelle heuristique permet des gains significatifs sur notre exemple d'application, que ce soit en nombre d'affectation (de nœuds) ou en temps CPU. Pour des raisons pratiques, l'étude de l'heuristique $dom/wdeg_{cbj}$ est limitée au cadre des CSPs binaires.

Nous proposons également, dans le cadre des CSPs Dynamiques (DCSP), un mécanisme d'apprentissage permettant d'exploiter l'effort de résolution des CSPs statiques successifs. Ce mécanisme d'apprentissage consiste à mémoriser les poids entre les CPSs successifs $P_{(0)}, P_{(1)}, \cdots, P_{(N)}$. L'idée principale est d'initialiser les poids du CPS $P_{(i)}$ par ceux du CPS $P_{(i-1)}$.

Les deux contributions académiques de cet article, l'heuristique dynamique de choix de variable et le mécanisme d'apprentissage pour les DCSP, ont été identifiées dans un context industriel, durant l'amélioration d'une méthode existante permettant de valider le design de la matrice de routage d'une charge utile télécom. Cette validation consiste à évaluer la tolérance d'une matrice à un nombre fixé de pannes de tubes amplificateurs TWTA (pour Travelling Wave Tube Amplifier). La propriété de tolérance est assurée par l'existence d'un mécanisme de redondance basé sur un ensemble R de tubes TWTA redondant et une matrice de routage permettant d'accéder aux tubes TWTA nominaux et redondant. Dans le cas de R pannes de TWTAs nominaux (utilisés dans des conditions nominales), la matrice de routage permet d'accéder aux R TWTAs redondant en considérant la configuration courante de la charge utile. Afin de valider le design de la matrice de routage et de fournir des statistiques sur les cas de pannes pour lesquels il n'y a pas d'accès possible aux TWTAs redondant via la matrice de routage, le schéma de validation choisie consiste en une énumération exhaustive de toutes les combinaisons, où une combinaison est définie par une configuration de la charge utile et un ensemble de R TWTAs en pannes. Une combinaison pour laquelle il n'y a pas d'accès possible aux TWTAs redondant via la matrice de routage, est dite *non valide*. Ces statistiques sont ensuite utilisés par les ingénieurs afin de modifier le design si besoin. Pour une matrice donnée, le nombre N de combinaisons peut être égale à plusieurs millions dans le meilleur des cas, voir plusieurs milliards dans le pire. La validation quant à elle peut nécessiter plusieurs heures dans le meilleur des cas, voir plusieurs jours dans le pire.

Soit $P_{(i)}$ un CSP binaire associé à une combinaison i, avec des contraintes de compatibilité entre les valeurs. Ainsi, valider le design d'une matrice de routage consiste à résoudre la séquence $P_{(0)}, P_{(1)}, \cdots, P_{(N)}$ de N CSPs, où chacun est le résultat d'un certain nombre de changements appliqués au précédent, ce qui correspond à la definition d'un Problème Dynamique de Satisfaction de Contraintes (DCSP). Ces changements sont des ajouts et/ou suppressions de valeurs, et activations et désactivations de variables (et/ou

de contraintes), qui correspondent respectivement à l'énumération des sous-combinaisons de R pannes de TWTAs et à l'énumération des sous-combinaisons de configuration de charge utile. Pour chaque configuration, T sous-combinaisons de R pannes de TWTAs sont générées, ainsi $N = PC * T$, où PC est le nombre de configurations de charge utile. Une combinaison dont le CSP associé admet au moins une solution est dite *valide*, *non-valide* dans le cas contraire.

La particularité de cette sequence de CSPs tient dans le fait que tous les T CPSs, une variable est activée et une autre est désactivée, et entre les CSPs successifs ayant les mêmes variables actives, des changements de valeurs sont effectués. Ainsi, deux CSPs séparés de $T-1$ CSPs ne diffèrent que d'une variable et de quelques valeurs. Il en résulte une forte similitude entre les CSPs successifs. Pour des raisons de confidentialité, nous ne somme pas autorisé à donner plus de details sur la modélisation. Cependant, cela ne nuira pas à la comprehension de l'article.

La méthode utilisée pour résoudre chaque CSP est la méthode hybride *FC-CBJ* associée à de l'Arc Cohérence (AC3.2bit [17]) en phase de prétraitement et à la nouvelle heuristique dirigée par les conflits, $dom/wdeg_{cbj}$, qui constitue notre premiere contribution. La seconde contribution est le mécanisme d'apprentissage des conflits entre les CSPs successifs. Cette combinaison de méthodes et de mécanismes fournit les meilleurs performances sur notre problématique de validation, notamment face à la méthode prospective Maintien de l'Arc Coherence (MAC).

Cet article est organisé comme suit. Dans un premier temps, nous donnons quelques définitions classiques dans le domaine des CSPs. Par la suite, nous décrivons plus en detail l'heuristique *dom/wdeg* proposée dans [11] et la nouvelle heuristique $dom/wdeg_{cbj}$ que nous proposons. Suite à quoi, nous présentons le mécanisme d'apprentissage appliqué à un DCSP binaire de T CSPs. Nous présentons ensuite les résultats d'expérimentations menées sur des instances CSPs structurées issues de la problématique industrielle de validation. Nous terminerons avec une conclusion et des perspetives.

2 Préliminaires

Dans cette section nous introduisons quelques définitions et notations utilisées dans la suite de l'article. Ces définitions et notations sont issues de [11].

Un Problème de Satisfaction de Contrainte est défini par un ensemble de variables prenant leurs valeurs dans un domaine et un ensemble de contraintes qui conditionnent les valeurs que pourront prendre les variables. Il est généralement présenté sous la forme d'un réseau de contraintes (V, C).

Définition 1 *Un réseau de contraintes est un couple (V, C) où :*

- *$V = \{V_1, \cdots, V_n\}$ est un ensemble fini de n variables. À chaque variable V_i correspond un domaine $dom(V_i)$ représentant l'ensemble des valeurs pouvant lui être affectée.*
- *$C = \{C_1, \cdots, C_m\}$ est un ensemble fini de m contraintes. Une contrainte $C_j \in C$ est associée à une relation $rel(C_j)$ représentant l'ensemble des tuples autorisés pour les variables $vars(C_j)$ liées par la contrainte C_j.*

Une contrainte C_j lie une variable V_i si et seulement si $V_i \in vars(C_j)$. Le nombre $|vars(C_j)|$ de variables liées par une contrainte C_j définit l'arité de celle-ci. Une contrainte binaire lie deux variables (arité égale à deux). En revanche, le nombre de contraintes liant une variable V_i représente le degré de cette dernière.

L'affectation d'une valeur à une variable est appelée instantiation (ou assignation). Soit $s = (v_1, v_2, \cdots, v_p)$ une instantiation partielle où p variable sont affectées, avec $p = |s| \leq |V|$ et v_i la valeur affectée à la variable V_i. Si $p = |V|$, s est appelée instantiation totale. Une solution est une instantiation totale de sorte que chaque contrainte soit satisfaite.

Un réseau de contraintes est aussi appelé instance CSP. Par abus de langage, CSP et instance CSP sont parfois confondus. Sans perte de généralité, dans cet article, L'ensemble $vars(C_j)$ des variables liées à la contrainte C_j est considéré comme étant ordonné. Il est alors possible d'obtenir la position $pos(V_i, C_j)$ d'une variable V_i dans $vars(C_j)$. Un tuple t est dit autorisé par une contrainte C_j si et seulement si $t \in rel(C)$. Un tuple t est dit support de l'instantiation (V_i, v) dans C_j si t est autorisé par C_j et tel que $t[pos(V_i, C_j)] = v$. (V_i, v) est dite consistant par rapport à C_j si et seulement si il existe un support de (V_i, v) dans C_j. L'opération consistant à déterminer si un tuplet est support d'une contrainte est appelé test de consistance

Définition 2 *Soient P une instance CSP, C_j une contrainte de P, V_i une variable de $vars(C_j)$ et v une valeur de $dom(V_i)$. (V_i, v) est dit consistant par rapport à C_j si et seulement si il existe un support de (V_i, v) dans C_j. C_j est dit arc-consistant si et seulement si pour toute variable V_i de $vars(C_j)$ et pour toute valeur v de $dom(V_i)$, (V_i, v) est consistant par rapport à C_j. P est arc-consistant si et seulement si toute contrainte de P est arc-consistante.*

La résolution d'un Problème de Satisfaction de Contrainte est la tâche de trouver une solution à une

instance CSP. Ce problème est NP-complet. Une instance CSP est dite satisfiable si et seulement si le réseau de contraintes associé admet au moins une solution. Dans le cas contraire, cette instance est dite non-satisfiable. Résoudre un CSP consiste alors, soit à déterminer sa non-satisfiabilité, soit à trouver au moins une solution.

3 Pondération des contraintes à partir des conflits pour une méthode prospective

L'un des points d'amélioration d'un algorithme de recherche arborescente de type retour arrière est l'ordre d'instantiation des variables, un ordre considéré comme crucial depuis l'avènement de ce type d'algorithme. Les heuristiques généralement utilisées sont *dom* [16], *dom/deg* [4], *dom/ddeg* [4, 24], $dom \oplus deg$ [12] et $dom \oplus ddeg$ [23, 5]. Ces heuristiques exploitent un certain nombre d'informations sur l'état courant de la recherche, telles que la taille courante des domaines (*dom*) ou le degré (courant) des variables (*deg* dans le cas statique et *ddeg* dans le cas dynamique). Aucune information sur l'état passé n'est exploitée.

En 2004, [11] proposent, dans un cadre général, celui des CSPs n-aire, une heuristique de choix de variable originale, *wdeg* (pour weighted degree), dirigée par les conflits. Cette heuristique dynamique est basée sur un poids *weight* associé à chaque contrainte du problème. Pour une contrainte C_j, le poids *weight* est incrémenté de un à chaque conflit impliquant cette dernière. L'évaluation du degré pondéré $\alpha_{wdeg}(V_i)$ d'une variable V_i est alors égale à la somme des poids des contraintes liant V_i et au moins une seconde variable non instantiée ; plus formellement :

- $\alpha_{wdeg}(V_i) = \sum_{V_i \in vars(C_j) \wedge |FutVars(C_j)|>1} weight[C_j]$

, où | $FutVars(C_j)$ | représente le nombre de variables non instantiées dans $vars(C_j)$. Dans la suite de l'article, le degré pondéré α_{wdeg} est noté *wdeg*. Dans le même article, les auteurs proposent l'heuristique *dom/wdeg*, qui consiste à sélectionner prioritairement la variable avec le plus petit rapport : taille courante du domaine de la variable, *dom*, sur le degré pondéré de cette dernière, *wdeg*. L'heuristique *dom/wdeg* compense l'absence d'un schéma rétrospectif (analyse des conflits) par un apprentissage lui permettant d'orienter la recherche vers les parties difficiles ou inconsistantes du problème.

Les auteurs ont proposé cette heuristique dans le cadre de méthodes prospectives tel que le *Forward-Checking* [16] (FC) et le *Maintien de l'Arc Coherence* [13, 21] (MAC). Ces méthodes mettent en œuvre des mécanismes de filtrage permettant de modifier le problème selon l'assignation réalisée, ceci grâce à des révisions successives des variables de manière à éliminer les valeurs devenues inconsistantes. Cette révision correspond par exemple à l'utilisation d'un filtrage gros grain (orienté arc pour cet exemple). Soit Q l'ensemble des arcs à reviser, un arc étant défini comme un couple *(Contrainte,Variable)*. La révision de l'arc (C_j, V_i) consiste à éliminer les valeurs de $dom(V_i)$ devenues inconsistantes avec C_j.

Algorithme 1 filtrer(Q : ensemble d'arcs) :booleen

```
1:  tantque Q ≠ ∅ faire
2:    Sélectionner et éliminer (C_j, V_i) de Q
3:    si reviser(C_j, V_i) alors
4:      si dom(V_i) = ∅ alors
5:        weight[C_j]++ // incrémentation du compteur de la contrainte
6:        retourner ECHEC
7:      sinon
8:        mettre à jour Q // en fonction de l'algorithme de filtrage
9:      finsi
10:   finsi
11: fin tantque
12: retourner SUCCES
```

La fonction *filter*, qui prend en paramètre l'ensemble Q, permet de reviser un certain nombre d'arcs itérativement (l'initialisation de Q est spécifique à FC et à MAC). Lorsqu'une impasse ($dom(V_i) = \emptyset$) est rencontrée lors d'une révision effective de l'arc (C_j, V_i), le poids *weight* associé à la contrainte C_j est incrémenté de un. Le filtrage se termine alors sur un échec. Dans le cas contraire, l'ensemble Q est mis à jour. La révision est effectuée par la fonction *reviser*.

Algorithme 2 reviser(C_j : contrainte, V_i : Variable) :booleen

```
nbElements ← |dom(V_i)|
pour chaque v ∈ dom(V_i) faire
  si rechercherSupport(C_j, V_i, v) = false alors
    retirer v de dom(V_i)
  finsi
fin pour
retourner nbElements ≠ |dom(V_i)|
```

Les auteurs ont montré l'efficacité d'une telle heuristique face aux heuristiques classiques sur un certain nombre de problèmes académiques structurés et non structurés. Cependant, cette heuristique n'a été présentée que pour des méthodes prospectives, dont le schéma d'exploration est du type retour arrière chronologique (backtrack). On peut donc s'interroger sur l'apport de la transposition d'une telle heuristique

pour des méthodes rétrospectives, voir hybrides, tel que la méthode hybride *FC-CBJ* alliant un schéma prospectif et un schéma rétrospectif, mis en œuvre respectivement par le *Forward-Checking* et le *Conflict Directed Backjumping*.

4 Heuristique dirigée par les conflits pour une méthode hybride : FC-CBJ

La méthode hybride *FC-CBJ* exploite la capacité de filtrage à moindre coût de la méthode prospective *Forward-Checking* et la capacité de retour arrière de la méthode rétrospective *Conflict Directed Backjumping* qui permet d'identifier la variable source de l'échec. Dans [11], l'heuristique *dom/wdeg* ne dispose pas de cette information (la variable source de l'échec) car elle est appliquée à un schéma de recherche de type retour arrière chronologique (backtrack). Appliquer l'heuristique *dom/wdeg* à la méthode *FC-CBJ* sans prendre en compte la capacité de cette dernière à identifier les variables sources de l'échec, risque d'orienter la recherche vers des parties non nécessairement difficiles ou inconsistantes du problème.

Nous proposons d'enrichir la phase d'apprentissage de l'heuristique *dom/wdeg* en redéfinissant le poids *weight* afin d'exploiter la capacité de la méthode retrospective *CBJ* à identifier la source de l'échec. Il en résulte une nouvelle heuristique dirigée par les conflits, $dom/wdeg_{cbj}$. Pour des raisons pratiques, l'étude de l'heuristique $dom/wdeg_{cbj}$ est limitée au cadre des CSPs binaires.

Tout d'abord, il est important de remarquer que dans un cadre binaire, il y a équivalence entre associer un compteur *weight* à chaque contrainte ou à chaque variable. Pour cela il suffit de remplacer la ligne 5 de l'algorithme 1 par les deux lignes suivantes :

- $weight[V_i][secondVar(C_j, V_i)] + +$
- $weight[secondVar(C_j, V_i)][V_i] + +$

C_j est la contrainte à l'origine d'une impasse lors de la révision de l'arc (V_i, C_j), et $secondVar(C_j, V_i)$ une fonction retournant la seconde variable liée par la contrainte C_j, la première étant la variable V_i (voir l'algorithme 3). Incrémenter le poids *weight* associé à une contrainte C_j d'arité égale à deux revient donc à incrémenter les poids *weight* associés aux deux variables liées par C_j. Le degré pondéré d'une variable V_i s'exprime alors de la façon suivante :

- $\alpha_{wdeg}(V_i) = \sum_{V_k \in V, V_k \in FutVars(V)} weight[V_i][V_k]$

Où $\mid FutVars(V) \mid$ représente l'ensemble des variables non instantiées appartenant à V.

Algorithme 3 filtrer(Q : ensemble d'arcs) :booleen

```
1: tantque Q ≠ ∅ faire
2:   Sélectionner et éliminer (C_j,V_i) de Q
3:   si reviser(C_j,V_i) alors
4:     si dom(V_i) = ∅ alors
5:       weight[V_i][secondVar(C_j, V_i)] + +
6:       weight[secondVar(C_j, V_i)][V_i] + +
7:       retourner ECHEC
8:     sinon
9:       mettre à jour Q
10:    finsi
11:  finsi
12: fin tantque
13: retourner SUCCES
```

Comme dit précédemment, afin d'exploiter la capacité du *CBJ* à identifier la source de l'échec, nous proposons d'enrichir la phase d'apprentissage de l'heuristique *dom/wdeg*. Pour cela, nous redéfinissons le mécanisme d'incrémentation du poids *weight*, qui est à présent nommé $weight_{cbj}$. Durant une recherche arborescente de type *FC-CBJ*, lors de l'échec de l'instantiation de la variable courante V_{curr} ($dom(V_{curr}) = \emptyset$ à cause des révisions successives de *FC*), la méthode rétrospective *CBJ* permet d'identifier la variable V_{jump} dont l'affectation courante est responsable de l'échec. La variable V_{jump} doit alors changer de valeur afin de débloquer la situation. L'idée est alors de conserver l'information d'échec entre les deux variables en incrémentant les poids $weight_{cbj}$ associés aux variables V_{curr} et V_{jump}. Le poids $weight_{cbj}$ nous permet ensuite d'évaluer le degré pondéré $\alpha_{wdeg_{cbj}}(V_i)$ de la variable V_i et ainsi de définir la nouvelle heuristique dynamique $dom/wdeg_{cbj}$, qui consiste à sélectionner prioritairement la variable avec le plus petit rapport : taille courante du domaine de la variable, *dom*, sur le degré pondéré $wdeg_{cbj}$ de cette dernière.

Dans le but de mettre en évidence l'intérêt de la redéfinition du mécanisme d'incrémentation du poids *weight* lors de l'adaptation de l'heuristique *dom/wdeg* à la méthode *FC-CBJ*, nous avions choisi d'associer les poids *weight* (et $weight_{cbj}$) aux variables (ce qui est équivalent à les associer aux contraintes dans le cas binaire). Cependant, il est possible d'adopter la définition de [11]. Pour cela, les explications d'échec doivent être maintenues sous forme d'ensemble de contraintes menant au conflit, ce qui permet de maintenir la notion de poids associé à chaque contrainte. En cas de conflit, il suffit alors d'incrémenter les poids des contraintes présentent dans le conflit. Ainsi, le degré pondéré d'une variable aurait la même définition que dans [11] et l'heuristique $dom/wdeg_{cbj}$ devient alors applicable dans un cadre n-aires.

5 Mémorisation de conflits pour un CSP Dynamique de T CSPs

Un CSP Dynamique (DCSP) a été introduit par [10] comme une suite de CSPs statiques $P_{(0)}, P_{(1)}, \cdots, P_{(N)}$, chacun résultant du précédent par des opérations sur les valeurs (ajout et/ou suppression). Dans [18], un DCSP est une généralisation d'un CSP [25], qui est défini par un ensemble de variables, un état d'activation des variables, un domaine pour chaque variable, et par deux types de contraintes : contraintes de compatibilité qui régissent l'association variable-valeur, et contraintes d'activation qui permettent d'activer ou de désactiver une variable selon l'assignation des autres variables. Une contrainte d'activation est de la forme : Si la valeur v est assignée à la variable V_i, alors $V_j, V_g, \cdot$ deviennent actives. Le terme *conditionnelle* serait peut être plus approprié.

Dans le cadre de la résolution de Problème Dynamique de Satisfaction de Contraintes[10, 18], de nombreux travaux ont été menés sur l'exploitation de l'effort de résolution des CSPs statiques successifs, notamment sur la réutilisation d'une solution précédemment calculée [1, 26], l'exploitation des explications de la non faisabilité [22] et l'adaptation des algorithmes d'arc consistance [3, 8]. Dans la même optique, nous proposons un mécanisme d'apprentissage par la mémorisation des conflits. En fonction de la similitude entre les CSPs successifs, une fois la recherche arborescente menée à terme, avec soit une preuve de la non-satisfiabilité, soit une preuve de satisfiabilité pour une instance CSP $P_{(i)}$, les poids $weight_{cbj}$ (or $weight$) permettent de mettre en évidence les parties difficiles ou inconsistantes de $P_{(i)}$, et potentiellement de $P_{(i+1)}$. Ainsi, durant la résolution de $P_{(i+1)}$, qui résulte d'un certain nombre de changements dans $P_{(i)}$, il serait intéressant de traiter en priorité les parties difficiles ou inconsistantes identifiées durant la résolution de $P_{(i)}$. Ceci permettrait, selon le principe *fail-first*, soit de détecter la non-satisfiabilité de l'instance plus tôt, soit d'accélérer la résolution en traitant en priorité les parties difficiles du problème (simplification des sous-problèmes inhérents). Dans la suite de cette section, ce qui est appliqué au poids $weight_{cbj}$ l'est également au poids $weight$.

Le mécanisme d'apprentissage est proposé dans le cadre d'un Problème Dynamique de Satisfaction de Contraintes (DCSP) de T CSPs, qui est une combinaison des deux définitions précédentes. Un DCSP de T CSPs est une séquence $P_{(0)}, P_{(1)}, \cdots, P_{(N)}$ de CSPs, chacun résultant du précédent par un certain nombre de changements. Ces changements sont des ajouts et/ou suppressions de valeurs, et activations et désactivations de variables. La particularité de cette sequence de CSPs tient dans le fait que tous les T CPSs, une variable est activée alors qu'une autre est désactivée, et entre les CSPs successifs ayant les mêmes variables actives, des changements de valeurs sont effectués. Ainsi, deux CSPs séparés de $T-1$ CSPs ne diffèrent que d'une variable et de quelques valeurs. Il en résulte une forte similitude entre les CSPs successifs.

Le mécanisme d'apprentissage consiste à initialiser le poids $weight_{cbj}$ de chaque variable du CSP $P_{(i+1)}$ par celui du CSP $P(i)$ ayant les mêmes variables. Quand deux CSPs successifs different d'une variable (et de quelques valeurs), les poids $weight_{cbj}$ sont réinitialisés à 1, ce qui permet de préserver le caractère discriminant des poids $weight_{cbj}$. Finalement, les poids $weight_{cbj}$ sont réinitialisés tout les T CSPs, d'où un DCSP de T CSPs. Il est important de noter que ce mécanisme est applicable à un DCSP quelconque. Dans la suite de l'article, l'association de la méthode *FC-CBJ*, de l'heuristique $dom/wdeg_{cbj}$ et du mécanisme d'apprentissage est notée $dom/wdeg^{T}_{cbj}$.

6 Exemple d'application et résultats

Comme mentionné précédemment, les contributions académiques de cet article on été identifiées dans un context industriel, durant l'amélioration d'une méthode existante permettant de valider le design de la matrice de routage d'une charge utile télécom. Cette validation consiste à résoudre une séquence $P_{(0)}, P_{(1)}, \cdots, P_{(N)}$ de CSPs binaires. Cette séquence résulte d'une énumération exhaustive de toutes les combinaisons, où une combinaison est définie par une configuration de la charge utile et un ensemble de R TWTAs en pannes. La séquence de CSPs binaires respecte la définition d'un DCSP de T CSPs binaire.

Afin de mettre en évidence l'intérêt pratique de ces contributions, nous avons conduit des expérimentations sur la problématique de validation (sur une machine Pentium Dual Core 2GHz 2Go , sous Windows XP). La nouvelle heuristique $dom/wdeg_{cbj}$ que nous proposons et le mécanisme d'apprentissage sont évalués selon le nombre de nœuds (#asgs) et le temps CPU de résolution (cpu, $hh:mm:ss$).

Dans un premier temps, nous comparons l'heuristique $dom/wdeg$ [11] et la nouvelle heuristique $dom/wdeg_{cbj}$ que nous proposons, toutes deux appliquées à la méthode de résolution précédemment décrite dans la section 1. Dans un second temps, nous évaluons l'apport de notre mécanisme d'apprentissage lors de la résolution des CSPs binaires successifs, dans le cadre d'un DCSP de T CSPs, où chaque CSP est résolu par la méthode décrite dans la section 1.

Avant de continuer, nous devons apporter une precision supplémentaire concernant le processus de va-

lidation. La problématique de validation est sujette à une contrainte supplémentaire qui limite le nombre de valeurs par variable. À la fin d'une validation, si des combinaisons non-valides ont été détectées, le processus de validation est relancé pour ces seules combinaisons, mais en incrémentant la borne supérieure de cette contrainte. Ainsi les nouvelles valeurs introduites permettant de diminuer le nombre de combinaisons non-valides. Le processus de validation est relancer tant que des combinaisons non-valides sont détectées, sous condition que la borne supérieure de la contrainte n'atteigne pas sa valeur maximale. Il faut noter que le processus de validation peut être lancé avec une borne supérieure égale à sa valeur maximale.

Les expérimentations sont menées sur trois types de DCSP, A, B et C, chacun correspondant à un design particulier de matrice de rouage. Soit B_i et C_i, avec $i = 1 \cdots 4$, des variantes, respectivement des DCSP B et C. Ces variantes résultent de l'ajout de valeurs inhérent à l'incrémentation de la borne supérieure de la contrainte précédemment exposée. Le tableau 1 illustre, par DCSP, le nombre de combinaisons, le nombre de combinaisons non-valides et les caractéristiques (nombre de variables, taille moyenne des domaines, un ordre de grandeur du nombre de contraintes) des CSPs associés aux combinaisons. Le DCSP B_1 a 10 829 485 combinaisons non-valides parmi 25 603 600 combinaisons. À la fin de la validation de B_1, les combinaisons non valides identifiées (10 829 485) vont composer le DCSP B_2, auquel des valeurs supplémentaires sont ajoutées du fait de l'incrémentation de la borne supérieure de la contrainte limitant le nombre de valeurs par variable. De la même manière, B_3 résulte de B_2, C_2 de C_1 et C_3 de C_2. Les DCSP B_4 et C_4 corresponde quant à eux, à une validation avec une borne supérieure égale à sa valeur maximale.

Le tableau 2 présente des statistiques en nombre d'assignation (#asgs) et de temps de validation (cpu) permettant de départager $dom/wdeg_{cbj}$ et $dom/wdeg$. On remarque, que pour l'ensemble des DCSPs, regroupant 86 126 235 de CSPs dont 26 594 967 non-satisfiables, l'heuristique $dom/wdeg_{cbj}$, que nous proposons, donne de meilleurs résultats en terme de nombre d'assignations et de temps cpu que l'heuristique $dom/wdeg$. La redéfinition des poids $weight$ a permis de d'exploiter, plus efficacement, la capacité de la méthode retrospective *CBJ* à identifier la source du conflit. Pour un coût d'implémentation limité, le nombre d'assignations et le temps cpu ont été respectivement réduits de 1,43% à 21,51% et de 0,66% à 17,61%. Ainsi, au lieu d'appliquer $dom/wdeg$ à al méthode hybride *FC-CBJ*, il est plus intéressant d'appliquer son extension, l'heuristique $dom/wdeg_{cbj}$.

Le tableau 3 illustre des statistiques en nombre d'assignations (#asgs) et de temps de validation (cpu) permettant d'évaluer l'apport du mécanisme d'apprentissage. On remarque une réduction significative de ces deux paramètres pour les DCSPs A_1, B_1, C_1, C_2 et C_4. Par exemple, le temps de validation du DCSP C_4 est égale à environ 13 heures avec le mécanisme d'apprentissage, alors qu'il est d'environ deux jours et demi sans le mécanisme d'apprentissage. Il est également important de remarquer que la validation des DCSPs B_3, B_4 et C_3 voient leurs temps cpu augmenter (au maximum de 9%) avec le mécanisme d'apprentissage. Néanmoins, ce temps reste inférieur à celui obtenu avec l'heuristique $dom/wdeg$ sans mémorisation. Ainsi, avec un coût d'implementation très faible, le mécanisme d'apprentissage offre des améliorations significatives sur notre exemple d'application.

7 Conclusion

Dans cet article, nous avons introduit une nouvelle heuristique dynamique de choix de variable dirigée par les conflits, l'heuristique $dom/wdeg_{cbj}$, appliquée à une méthode hybride *FC-CBJ*, alliant un schéma prospectif et un schéma rétrospectif, mis en œuvre respectivement par le *Forward-Checking* et le *Conflict Directed Backjumping*. Nous avons également proposé un mécanisme d'apprentissage permettant d'exploiter l'effort de résolution des CSPs statiques successifs dans le cadre d'un DCSP de T CSPs. Ce mécanisme, simple et peu coûteux, repose sur la mémorisation des compteurs $weight_{cbj}$ entre les CSPs successifs. L'heuristique $dom/wdeg_{cbj}$ a été fortement inspirée par l'heuristique $dom/wdeg$ [11] de Boussemart et al. Néanmoins, $dom/wdeg_{cbj}$ se distingue de $dom/wdeg$ par un apprentissage par les conflits plus adapté à la méthode rétrospective *CBJ*, ce qui lui permet de la surclasser en terme de nombre de nœuds et de temps CPU sur notre exemple d'application. Les expérimentations menées sur cet exemple d'application ont permis de mettre en évidence l'apport significatif du mécanisme d'apprentissage, notamment sur le design C4 qui est actuellement utilisé pour l'élaboration des nouvelles matrices de routage.

L'heuristique $dom/wdeg_{cbj}$ et le mécanisme d'apprentissage ont été proposé dans un cadre bien particulier, celui des CSPs binaires et des DCSP de T CSPs. Néanmoins, comme dit précédemment, leurs transcriptions dans un cadre plus général, celui des CSPs n-aire et des DCSPs quelconque est possible. De plus, le mécanisme d'apprentissage peut être étendu à d'autres méthodes de résolution, autre que la méthode hybride *FC-CBJ*.

Remerciements

Nous tenons à remercier Christophe Lecoutre pour avoir mené des expérimentations sur un échantillon de combinaisons et de nous avoir orienté vers l'heuristique *dom/wdeg*.

Références

[1] J. Maloney B. N. Freeman-Benson and A. Borning. An incremental constraint solver. *Communications of the ACM*, 33(1) :54–63, 1990.

[2] F. Bacchus. Extending forward checking. In *Proceedings of the CP'00*, pages 35–51, 2000.

[3] C. Bessière. Arc-Consistency in Dynamic Constraint Satisfaction Problems. In *Proceeding of The Ninth National Conference on Artificial Intelligence*, pages 221–226, 1991.

[4] C. Bessiere and J. Regin. MAC and combined heuristics : two reasons to forsake FC (and CBJ ?) on hard problems. In *Proceedings of the CP'96*, pages 61–75, 1996.

[5] D. Brelaz. New methods to color the vertices of a graph. *Communications of the ACM*, 22 :251–256, 1979.

[6] F. Boussemart C. Lecoutre and F. Hemery. Techniques de retour-arrière intelligent versus heuristiques dirigées par les conflits. In *Proceedings of JNPC'04*, 2004.

[7] A. Chmeiss and L. Saïs. De FC à MAC : un algorithme paramétrable pour la résolution des CSP. pages 267–276, Lens, 2005.

[8] R. Debruyne. Les algorithmes d'arc-consistence dans les csp dynamiques. *Revue d'intelligence artificielle*, 9(3) :239–267, 1995.

[9] R. Dechter and D. Frost. Backjump-based backtracking for constraint satisfaction problems. *Artificial Intelligence*, 136 :147–188, 2002.

[10] R. Detcher and A. Detcher. Belief maintenance in dynamic constraint networks. In *Proceedings of the 7th National Conference on Artifical intelligence*, pages 37–42, St. Paul, MN, USA, 1988.

[11] C. Lecoutre F. Boussemart, F. Hemery and L. Sais. Boosting systematic search by weighting constraints. In *Proceedings of the ECAI'O4*, pages 146–150, 2004.

[12] D. Frost and R. Dechter. Look-ahead value ordering for constraint satisfaction problems. In *Proceedings of IJCAI'95*, pages 572–578, 1995.

[13] J. Gaschnig. Performance measurement and analysis of certain search algorithms. Technical report CMU-CS-79-124, Carnegie-Mellon, 1979.

[14] I. P. Gent and I. P. Underwood. The logic of search algorithms : Theory and applications. In *Proceedings of the Third International Conference on Principles and Practice of Constraint Programming (CP'97)*, pages 77–91, 1997.

[15] M. Ginsberg. Dynamic backtracking. *Artificial Intelligence*, 1 :25–46, 1993.

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

[17] C. Lecoutre and J. Vion. Enforcing arc consistency using bitwise operations. *Constraint Programming Letters(CPL)*, 2 :21–35, 2008.

[18] S. Mittal and B. Falkenhainer. Dynamic constraint satisfaction problems. In *Proceedings of the 8th National Conference on Artifical intelligence*, pages 25–32, 1990.

[19] R. Debruyne N. Jussien and P. Boizumault. Maintaining arc-consistency within dynamic backtracking. In *Proceedings of the CP'00*, pages 249–261, 2000.

[20] P. Prosser. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, 9(3) :268–299, August 1993.

[21] D. Sabin and E. Freuder. Contradicting conventional wisdom in constraint satisfaction. In *Proceedings of the PPCPA'94*, Seattle WA, 1994.

[22] T. Schiex and G. Verfaillie. Nogood recording for static and dynamic constraint satisfaction problems. *IJAIT*, 3(2) :187–207, 1994.

[23] B.M. Smith. The brelaz heuristic and optimal static orderings. In *Proceedings of CP'99*, pages 405–418, Alexandria, VA, 1999.

[24] B.M. Smith and S.A. Grant. Trying harder to fail first. In *Proceedings of ECAI'98*, pages 249–253, Brighton, UK, 1998.

[25] E. Tsang. *Foundations of Constraint Satisfaction.* Academic Press Limited, Sandiego, 1993.

[26] G. Verfaillie and T. Schiex. Solution reuse in dynamic constraint satisfaction problems. In *Proceeding of the Twelfth National Conference on Artificial Intelligence*, pages 307–312, 1994.

DCSPs	Nombre de combinaisons		nbVar	nbVal	nbContraintes
	Total	Invalide		(moyenne)	($\approx$)
A_1	11 625 120	240	22	8	$\approx$ 900
B_1	25 603 600	10 829 485	21	10	$\approx$ 2400
B_2	10 829 485	8 695 364	21	14	$\approx$ 8000
B_3	8 695 364	3 126 532	21	22	$\approx$ 28000
B_4	25 603 600	3 126 532	21	22	$\approx$ 28000
C_1	7 312 032	475 768	25	11	$\approx$ 4500
C_2	475 768	294 354	25	19	$\approx$ 21000
C_3	294 354	23 466	25	34	$\approx$ 87000
C_4	7 312 032	23 466	25	34	$\approx$ 87000

TABLE 1 – Données des diffèrent DCSPs

DCSPs		$dom/wdeg$	$dom/wdeg_{cbj}$	$dom/wdeg_{cbj}$ vs $dom/wdeg$
A_1	#asgs	556 791 037	548 839 044	**-1,43%**
	cpu	00 :58 :14	00 :57 :51	**-0,66%**
B_1	#asgs	1 854 403 849	1 628 697 906	**-12,17%**
	cpu	03 :45 :21	03 :30 :57	**-6,39%**
B_2	#asgs	8 444 489 777	7 644 525 648	**-9,47%**
	cpu	12 :03 :07	11 :02 :34	**-8,37%**
B_3	#asgs	19 335 093 642	17 250 317 028	**-10,78%**
	cpu	28 :41 :01	26 :05 :21	**-9,05%**
B_4	#asgs	21 059 047 647	18 860 453 983	**-10,44%**
	cpu	34 :11 :17	31 :24 :56	**-8,11%**
C_1	#asgs	745 606 878	696 215 827	**-6,62%**
	cpu	02 :02 :07	01 :58 :34	**-2,91%**
C_2	#asgs	2 696 933 390	2 116 805 967	**-21,51%**
	cpu	04 :03 :51	03 :20 :54	**-17,61%**
C_3	#asgs	5 136 698 506	4 776 536 475	**-7,01%**
	cpu	09 :26 :42	08 :54 :25	**-5,70%**
C_4	#asgs	22 188 614 328	21 146 498 726	**-4,70%**
	cpu	56 :36 :03	56 :20 :16	**-0,46%**

TABLE 2 – $dom/wdeg_{cbj}$ vs $dom/wdeg$

DCSPs		$dom/wdeg_{cbj}$	$dom/wdeg^T_{cbj}$	$dom/wdeg^T_{cbj}$ vs $dom/wdeg_{cbj}$
A_1	#asgs	548 839 044	320 962 468	**-41,52%**
	cpu	00 :57 :51	00 :45 :59	**-20,51%**
B_1	#asgs	1 628 697 906	1 365 322 208	**-16,17%**
	cpu	3 :30 :57	3 :12 :19	**-8,83%**
B_2	#asgs	7 644 525 648	7 398 902 997	**-3,21%**
	cpu	11 :02 :34	10 :22 :58	**-5,98%**
B_3	#asgs	17 250 317 028	18 829 599 249	9,16%
	cpu	26 :05 :21	28 :01 :24	7,41%
B_4	#asgs	18 860 453 983	20 386 953 037	8,09%
	cpu	31 :24 :56	33 :19 :55	6,10%
C_1	#asgs	696 215 827	227 762 909	**-67,29%**
	cpu	1 :58 :34	1 :12 :06	**-39,19%**
C_2	#asgs	2 116 805 967	1 739 005 574	**-17,85%**
	cpu	3 :20 :54	2 :45 :34	**-17,59%**
C_3	#asgs	4 776 536 475	4 802 963 000	0,55%
	cpu	8 :54 :25	8 :58 :31	0,77%
C_4	#asgs	21 146 498 726	5 802 664 153	**-72,56%**
	cpu	56 :20 :16	13 :36 :51	**-75,83%**

TABLE 3 – Mécanisme d'apprentissage

Backtracking asynchrone agile pour les problèmes de satisfaction de contraintes distribués

Christian Bessiere[1] **El Houssine Bouyakhf**[2] **Younes Mechqrane**[2] **Mohamed Wahbi**[1,2]

[1]LIRMM/CNRS, University Montpellier 2, France
[2]LIMIARF/FSR, University Mohammed-V Agdal, Morroco
{bessiere,wahbi}@lirmm.fr ymechqrane@spsm.ma bouyakhf@fsr.ac.ma

Résumé

Le Backtracking Asynchrone est la procédure de recherche standard pour le raisonnement par contraintes distribué. Cette procédure nécessite un ordre total sur les agents. Tous les algorithmes avec une complexité spatiale polynomiale proposés jusqu'à présent pour améliorer le Backtracking Asynchrone ne permettent de réordonner les agents que de manière restreinte. Dans ce papier, nous proposons Agile-ABT, une procédure de recherche qui permet de réordonner les agents avec plus de liberté que les approches précédentes. Cette flexibilité est due à la notion de *valeurs de terminaison*, qui sont des vecteurs d'entiers positifs associés aux ordres échangés par les agents au cours de la recherche. Dans Agile-ABT, les agents peuvent proposer de nouveaux ordres autant qu'ils veulent à condition que les valeurs de terminaison décroissent au fur et à mesure que la recherche progresse. Nos expériences montrent les bonnes performances de Agile-ABT en comparaison avec les autres algorithmes de changement d'ordre.

1 Introduction

Des problèmes divers dans l'intelligence artificielle distribuée peuvent être formalisés sous forme de *Problèmes de Satisfaction de Contraintes Distribués* (DisCSP). Ces problèmes consistent à trouver une combinaison cohérente des actions de plusieurs agents (ex, allocation distribuée de ressources [6], réseaux de capteurs [1]). Les DisCSP sont composés d'agents, chacun ayant localement son propre réseau de contraintes. Les variables des différents agents sont liées par des contraintes. Les agents doivent affecter des valeurs à leurs variables de telle sorte que toutes les contraintes soient satisfaites.

Plusieurs algorithmes distribués ont été développés pour résoudre les DisCSP. Le backtracking asynchrone (ABT) est un des principaux.

ABT est un algorithme asynchrone exécuté de manière autonome par chaque agent où les agents n'ont pas à attendre les décisions des autres. Néanmoins, les agents sont soumis à un ordre total (priorité). Chaque agent essaie de trouver une affectation qui soit consistante avec les choix des agents les plus prioritaires. Quand un agent affecte une valeur à sa variable, la valeur sélectionnée est envoyée aux agents moins prioritaires. Lorsqu'un agent ne trouve aucune valeur consistante, l'incohérence est signalée aux agents les plus prioritaires par le biais d'un nogood (une combinaison de valeurs infructueuses). ABT calcule une solution (ou détecte qu'il n'y en a pas) en un temps fini. L'ordre total sur les agents est statique.

Il est connu des CSP centralisés que le fait de modifier l'ordre des variables d'une manière dynamique durant la recherche peut accélérer d'une manière drastique la recherche. Dans ce cadre, plusieurs algorithmes ont été proposés.

Asynchronous Weak Commitment (AWC) réordonne les agents d'une manière dynamique au cours de la recherche en plaçant l'émetteur d'un nogood plus haut dans l'ordre que les autres agents impliqués dans le nogood [9]. Cependant AWC a une complexité spatiale exponentielle.

Silaghi et al. (2001) ont présenté une tentative d'hybridation entre ABT et AWC où les changements d'ordre sont effectués par des agents abstraits. Dans, [7] l'heuristique du backtracking dynamique centralisé a été appliquée à ABT. Toutefois, dans les deux études précitées, les améliorations obtenues par rapport à ABT étaient mineures.

Zivan et Meisels (2006) ont proposé un algorithme de changement d'ordre pour le backtracking asynchrone

(ABTDO). Dans ABTDO, quand un agent affecte une valeur à sa variable, il peut modifier l'ordre des agents moins prioritaires.

Un nouveau type d'heuristiques pour le changement d'ordre, appelées heuristiques rétroactives, est présenté dans [12]. Dans la meilleure de ces heuristiques, l'agent qui a généré le nogood doit être placé entre le dernier et l'avant dernier agent dans le nogood. Cependant, la position du générateur du nogood ne sera effectivement modifiée que si la taille de son domaine est plus petite que les tailles des domaines des agents situés entre sa position actuelle et sa position cible.

Dans ce papier, nous proposons Agile-ABT, un algorithme de changement d'ordre asynchrone qui s'affranchit des restrictions habituelles des algorithmes de backtracking asynchrone. L'ordre des agents plus prioritaires que l'agent qui reçoit le message backtrack peut être changé avec plus de liberté tout en garantissant une complexité spatiale polynomiale. En plus, l'agent qui reçoit le message backtrack, appelé la *cible* du backtracking, n'est pas nécessairement l'agent ayant la priorité la plus faible parmi les agents conflictuels.

Le principe de Agile-ABT est construit sur les valeurs de terminaison échangées par les agents au cours de la recherche. Une valeur de terminaison est un tuple d'entiers positifs attaché à un ordre. Chaque entier positif dans ce tuple représente la taille présumée du domaine d'un agent ; la position de cet agent dans l'ordre étant la même que la position de l'entier dans le tuple. De nouveaux ordres sont proposés par les agents, sans aucun contrôle global, de manière à ce que les valeurs de terminaison décroissent lexicographiquement au fur et à mesure que la recherche progresse. Puisque une taille de domaine ne peut pas être négative, les valeurs de terminaison ne peuvent pas décroitre indéfiniment.

Quand un agent propose un nouvel ordre, il informe les autres en leur envoyant son nouvel ordre et sa nouvelle valeur de terminaison. Quand un agent compare deux ordres contradictoires, il va garder l'ordre qui est associé à la plus petite valeur de terminaison.

Le reste de ce papier est organisé comme suit. La section 2 rappelle quelques définitions de base. La section 3 décrit les concepts de base nécessaires pour sélectionner des nouveaux ordres qui font décroitre la valeur de terminaison. Nous présentons en détail notre algorithme dans la section 4. Nous apportons la preuve que cet algorithme est correct dans la section 5. Une étude expérimentale approfondie est donnée en section 6. La section 7 conclut le papier.

2 Préliminaires

Le Problème de Satisfaction de Contraintes Distribué a été formalisé dans [10] comme étant un tuple $(\mathcal{A}, \mathcal{X}, \mathcal{D}, \mathcal{C})$, où $\mathcal{A}$ est un ensemble d'agents $\{A_1, \ldots, A_a\}$, $\mathcal{X}$ est un ensemble de variables $\{x_1, \ldots, x_n\}$, tel que chaque variable x_i est contrôlée par un seul agent dans $\mathcal{A}$. $\mathcal{D} = \{D_1, \ldots, D_n\}$ est un ensemble de domaines, où D_i est un ensemble fini de valeurs qui peuvent être affectées à la variable x_i. La taille du domaine initial de la variable x_i est notée d_i^0. $\mathcal{C}$ est un ensemble de contraintes binaires qui spécifient les combinaisons de valeurs autorisées pour les couples de variables qu'elles impliquent. Une contrainte $c_{ik} \in \mathcal{C}$ entre deux variables x_i et x_k est un sous-ensemble du produit cartésien $D_i \times D_k$.

Pour des raisons de simplicité, nous considérons une version restreinte de DisCSP où chaque agent contrôle exactement une seule variable. Nous utilisons les termes agent et variable d'une manière interchangeable et nous identifions l'index d'un agent par l'indice de sa variable.

Chaque agent maintient son propre compteur et l'incrémente chaque fois qu'il change sa valeur. La valeur courante du compteur sera associée à chaque nouvelle affectation.

Définition 1 *Une **affectation** pour un agent $A_i \in \mathcal{A}$ est un tuple (x_i, v_i, t_i), où v_i est une valeur du domaine de x_i et t_i la valeur du compteur de A_i. Quand on compare deux affectations, la plus récente est celle associée avec le plus grand compteur t_i. Deux ensembles d'affectations $\{(x_{i_1}, v_{i_1}, t_{i_1}), \ldots, (x_{i_k}, v_{i_k}, t_{i_k})\}$ et $\{(x_{j_1}, v_{j_1}, t_{j_1}), \ldots, (x_{j_q}, v_{j_q}, t_{j_q})\}$ sont **cohérents** si chaque variable commune à ces deux sous ensembles est affectée à la même valeur.*

A_i est autorisé à mémoriser un ordre unique noté o_i. Les agents qui apparaissent avant A_i dans o_i sont les agents les plus prioritaires (prédécesseurs) notés par $Pred(A_i)$ et inversement les agents moins prioritaires (successeurs) $Succ(A_i)$ sont les agents qui apparaissent après A_i.

Définition 2 *La **Vue** (AgentView) d'un agent A_i est un tableau qui contient les affectations les plus récentes de $Pred(A_i)$.*

Les agents peuvent inférer des ensembles d'affectations inconsistants appelés ***nogoods*** . Un nogood peut être représenté sous forme d'une implication. Évidemment, il y a plusieurs possibilités pour représenter un nogood donné sous forme d'une implication. Par exemple, $\neg[(x_1 = v_1) \wedge \cdots \wedge (x_k = v_k)]$ est logiquement équivalent à $[(x_2 = v_2) \wedge \cdots \wedge (x_k = v_k)] \rightarrow (x_1 \neq v_1)$. Quand un nogood est représenté sous forme d'une implication, la partie gauche (lhs) et la partie droite (rhs) sont définies à partir de la position de $\rightarrow$. La partie droite d'un nogood est également appelée *conclusion* de ce nogood. Un nogood (ng) est ***compatible*** avec un ordre o_i si tous les agents dans $lhs(ng)$ apparaissent avant $rhs(ng)$ dans o_i.

Le domaine courant de x_i est l'ensemble de valeurs $v \in D_i$ telles que $x_i \neq v$ n'apparait dans aucune conclusion

de nogood mémorisé par A_i. Un agent maintient un seul nogood par valeur supprimée. La taille du domaine courant de A_i est notée par d_i.

3 Quelques notions et fonctions clés

Avant de présenter Agile-ABT, nous avons besoin d'introduire quelques nouvelles notions et de présenter quelques fonctions clés.

3.1 Établir une priorité entre les ordres

Quand les agents sont autorisés à proposer de nouveaux ordres d'une manière asynchrone, il faut que ces agents soient en mesure de décider d'une manière cohérente quel ordre sélectionner.

Nous proposons que la priorité entre les ordres soit basée sur les *valeurs de terminaison*. D'une manière informelle, si $o_i = [A_1, \ldots, A_n]$ est l'ordre courant de A_i, le tuple des tailles des domaines $[d_1, \ldots, d_n]$ est la valeur de terminaison associée à o_i. Pour construire les valeurs de terminaison, les agents échangent des *explications*.

Définition 3 *Une **explication** e_j est une assertion sous forme $lhs(e_j) \rightarrow d_j$, où $lhs(e_j)$ est la conjonction des parties gauches de tous les nogoods mémorisés par A_j, et d_j est le nombre de valeurs dans le domaine de A_j non supprimées par les nogoods. d_j est notée par $rhs(e_j)$.*

Chaque fois qu'un agent communique son affectation aux autres agents (en leur envoyant un message **ok?**), il insère son explication dans le message **ok?** afin de permettre aux autres agents de construire leurs valeurs de terminaison.

Les variables dans la partie gauche d'une explication e_j doivent précéder la variable x_j dans l'ordre puisque les affectations de ces variables ont été utilisées pour éliminer des valeurs du domaine de x_j. Une explication e_j induit certaines contraintes d'ordre appelées *safety conditions* dans [4].

Définition 4 ***Une contrainte d'ordre** est une assertion $x_k \prec x_j$. Étant donnée une explication e_j, $S(e_j)$ est l'ensemble des contraintes d'ordre d'ordre induites par e_j, avec $S(e_j) = \{(x_k \prec x_j) \mid x_k \in lhs(e_j)\}$.*

Une explication e_j est ***compatible*** avec un ordre o si toutes les variables dans $lhs(e_j)$ apparaissent avant x_j dans o. Chaque agent A_i mémorise un ensemble E_i d'explications envoyées par les autres agents. Durant la recherche, E_i est mis à jour afin de supprimer les explications qui ne sont plus valides.

Définition 5 *Pour un agent A_i, une explication e_j dans E_i est **valide** si elle est* compatible *avec l'ordre courant o_i et $lhs(e_j)$ est cohérent avec la Vue de A_i.*

Quand E_i contient une explication e_j (associée à A_j), A_i utilise cette explication pour déduire la taille du domaine courant de A_j. Sinon, A_i suppose que la taille du domaine courant de A_j est égale à d^0_j. La valeur de terminaison dépend de l'ordre et de l'ensemble des explications.

Définition 6 *Soit E_i un ensemble d'explications mémorisées par A_i, o un ordre sur les agents tel que chaque explication dans E_i est compatible avec o, et $o(k)$ telle que $A_{o(k)}$ est le $k^{ème}$ agent dans o. La **valeur de terminaison** $TV(E_i, o)$ est un tuple $[tv^1, \ldots, tv^n]$, où $tv^k = rhs(e_{o(k)})$ si $e_{o(k)} \in E_i$, sinon, $tv^k = d^0_{o(k)}$.*

Dans Agile-ABT, un ordre est toujours associé à une valeur de terminaison. Quand on compare deux ordres, le *plus fort* entre les deux est celui associé à la valeur de terminaison lexicographiquement inférieure. En cas d'égalité, on utilise l'ordre lexicographique sur les indexes des agents, le plus petit étant le plus fort.

3.2 La cible du backtracking

Lorsque toutes les valeurs d'un agent A_i sont supprimées par des nogoods, cet agent procède à la résolution de ses nogoods. Cette résolution produit un nouveau nogood ng qui est la conjonction des parties gauches de tous les nogoods mémorisés par A_i. Si ng est vide, alors l'inconsistance est prouvée. Sinon, un des agents impliqués dans le conflit doit changer sa valeur. Pour les algorithmes de backtracking asynchrone standards, l'agent ayant la plus basse priorité doit changer sa valeur. Agile-ABT s'affranchit de cette restriction en autorisant A_i à sélectionner avec plus de liberté la cible du backtracking. La seule restriction pour placer une variable x_k dans la conclusion de ng est de trouver un ordre o' tel que $TV(up\_E, o')$ est lexicographiquement inférieur à la valeur de terminaison associée à l'ordre courant de A_i. up_E est obtenu en mettant à jour E_i après avoir mis x_k dans la conclusion de ng.

La fonction `updateExplanations` prend en argument l'ensemble E_i, le nogood ng et la variable x_k à placer dans la conclusion de ng. `updateExplanations` supprime toutes les explications qui ne sont plus cohérentes une fois qu'on a placé x_k dans la conclusion de ng. Elle met à jour l'explication de l'agent A_k mémorisée par A_i et retourne un ensemble d'explications up_E.

Cette fonction ne crée pas de cycles dans l'ensemble des contraintes d'ordre $S(up\_E)$ si $S(E_i)$ est acyclique. En effet, toutes les explications ajoutées ou supprimées de $S(E_i)$ pour obtenir $S(up\_E)$ contiennent x_k. Donc, si $S(up\_E)$ contient des cycles, tous ces cycles doivent contenir x_k. Toutefois, il n'y aucune contrainte d'ordre sous forme $x_k \prec x_j$ dans $S(up\_E)$ puisque toutes ces contraintes d'ordre ont été supprimées dans la ligne 2. Par conséquent, $S(up\_E)$ ne peut pas être cyclique.

function `updateExplanations(`E_i, ng, x_k`)`

1. $up\_E \leftarrow E_i$; `setRhs(`*ng*, x_k`)`;
2. remove each $e_j \in up\_E$ such that $x_k \in$ `lhs(`e_j`)`;
3. **if** $(e_k \notin up\_E)$ **then**
4. `setLhs(`e_k,$\emptyset$`)`;
5. `setRhs(`e_k,d_k^0`)`;
6. add e_k to up_E ;
7. `setLhs(`e'_k, `lhs(`e_k`)` $\cup$ `lhs(`*ng*`))`;
8. `setRhs(`e'_k, `rhs(`e_k`)`-1`)`;
9. replace e_k by e'_k ;
10. `return(`up_E`)` ;

Comme nous allons le montrer dans la section 4, les mises à jour effectuées par A_i garantissent que $S(E_i)$ reste toujours acyclique. Par conséquent, $S(up\_E)$ est aussi acyclique et peut être représenté par un graphe acyclique dirigé $G = (N, U)$ tel que $N = \{x_1, \ldots, x_n\}$ est l'ensemble des noeuds et U est l'ensemble des arcs dirigés. Un arc $(j, l) \in U$ si et seulement si $(x_j \prec x_l) \in S(up\_E)$.

3.3 Comment faire décroitre les valeurs de terminaison ?

La terminaison de Agile-ABT est basée sur le fait que les valeurs de terminaison associées aux ordres sélectionnés par les agents décroissent au fur et mesure que la recherche progresse. Pour accélérer la recherche, Agile-ABT est écrit de façon à ce que les agents fassent décroitre les valeurs de terminaison autant qu'ils le peuvent. Quand un agent procède à la résolution de ses nogoods, il vérifie s'il peut trouver un nouvel ordre sur les agents de manière à ce que la valeur de terminaison associée à ce nouvel ordre soit meilleure que celle associée à l'ordre courant. Si c'est le cas, l'agent en question remplace son ordre courant et sa valeur de terminaison par ceux qu'il vient de calculer, et informe tous les autres agents.

Supposons maintenant qu'après avoir procédé à la résolution de ses nogoods, un agent A_i décide de placer x_k dans la conclusion du nogood ng produit par la résolution et soit up_E =`updateExplanations` (E_i, ng, x_k). La fonction `computeOrder` prend comme paramètre l'ensemble up_E et retourne un ordre up_o compatible avec l'ordre partiel induit par up_E.

Soit G le graphe acyclique dirigé associé à up_E. La fonction `computeOrder` détermine, à chaque itération p, l'ensemble $Roots$ des noeuds qui n'ont plus de prédécesseurs. Étant donné qu'on cherche à minimiser la valeur de terminaison, la fonction `computeOrder` sélectionne dans $Roots$ le noeud x_j ayant le plus petit domaine. Ce noeud est placé dans la $p^{ème}$ position et supprimé de G. Enfin, p est incrémenté et tous les arcs sortants qui partent de x_j sont supprimés de G.

Ayant proposé un algorithme qui détermine un ordre avec une petite valeur de terminaison pour une cible de backtracking x_k donnée, on a besoin de savoir comment

function `computeOrder(`up_E`)`

11. $G = (N, U)$ is the acyclic graph associated to up_E ;
12. $p \leftarrow 1$; o is an array of length n;
13. **while** $G \neq \emptyset$ **do**
14. $Roots \leftarrow \{x_j \in N \mid x_j$ has no incoming edges$\}$;
15. $o[p] \leftarrow x_j$ such that $d_j = min\{d_k \mid x_k \in Roots\}$;
16. `remove` x_j from G ; $p \leftarrow p + 1$;
17. `return(`o`)` ;

choisir x_k pour obtenir un ordre qui fera décroitre davantage la valeur de terminaison.

La fonction `chooseVariableOrder` teste toutes les variables x_k contenues dans le nogood, calcule le nouvel ordre et la valeur de terminaison pour chaque variable testée x_k (lignes 20 et 22), et renvoie l'information correspondant à l'ordre le plus fort.

function `chooseVariableOrder(`E_i, ng`)`

18. $o' \leftarrow o_i$; $TV' \leftarrow TV_i$; $E' \leftarrow nil$; $x' \leftarrow nil$;
19. **for each** $x_k \in ng$ **do**
20. $up\_E \leftarrow$ `updateExplanations(`E_i, *ng*, x_k`)`;
21. $up\_o \leftarrow$`computeOrder(`up_E`)`;
22. $up\_TV \leftarrow$ TV(up_E, up_o);
23. **if** (up_TV *is* `smaller than` TV') **then**
24. $x' \leftarrow x_k$;
25. $o' \leftarrow up\_o$;
26. $TV' \leftarrow up\_TV$;
27. $E' \leftarrow up\_E$;
28. `return(`$\langle x', o', TV', E' \rangle$`)` ;

4 L'algorithme

Chaque agent A_i mémorise localement une certaine quantité d'informations sur l'état global de la recherche, à savoir une Vue, un NogoodStore qui contient l'ensemble des nogoods mémorisés par A_i, un ensemble d'explications (E_i), un ordre courant (o_i) et la valeur de terminaison (TV_i). Agile-ABT autorise les types de messages suivants (A_i étant l'émetteur) :

- Un message **ok?** est envoyé par A_i aux agents moins prioritaires pour demander si la valeur qu'il a choisie est acceptable. En plus de la valeur choisie, le message **ok?** contient une explication e_i qui indique la taille du domaine courant de A_i. Le message **ok?** contient également l'ordre courant o_i et la valeur de terminaison courante TV_i mémorisés par A_i.
- Un message **ngd** est envoyé par A_i lorsque toutes ses valeurs sont supprimées par les nogoods contenus dans NogoodStore. Ce message contient le nogood proprement dit, ainsi que o_i et TV_i.
- Un message **order** sert à proposer un nouvel ordre. Ce message contient l'ordre o_i proposé par A_i et la valeur de terminaison TV_i.

Agile-ABT (Figures 1 et 2) est exécuté par chaque agent. Après l'initialisation, chaque agent affecte une valeur à sa

```
   procedure Agile-ABT ( )
29.  t_i ← 0 ; TV_i ← [∞, ∞, ..., ∞] ; end ← false ; v_i ← empty;
30.  CheckAgentView( ) ;
31.  while (¬end) do
32.     msg ← getMsg( );
33.     switch (msg.type) do
34.        ok?    : ProcessInfo(msg);
35.        order  : ProcessOrder(msg);
36.        ngd    : ResolveConflict(msg);
37.        stp    : end ← true ;

   procedure ProcessInfo(msg)
38.  CheckOrder( msg.Order, msg.TV ) ;
39.  UpdateAgentView( msg.Assig ∪ lhs( msg.Exp ) ) ;
40.  if msg.Exp is valid then add( msg.Exp, E );
41.  CheckAgentView( ) ;

   procedure ProcessOrder(msg)
42.  CheckOrder( msg.Order,msg.TV ) ;
43.  CheckAgentView( ) ;

   procedure ResolveConflict(msg)
44.  CheckOrder( msg.Order,msg.TV ) ;
45.  UpdateAgentView( msg.Assig ∪ lhs( msg.Nogood ) ) ;
46.  if Coherent( msg.Nogood, AgentView ∪ x_i = v_i ) and
     Compatible( msg.Nogood, o_i ) then
47.     add( msg.Nogood,NogoodStore ) ; v_i ← empty;
48.     CheckAgentView( );
49.  else if rhs( msg.Nogood ) =v_i then
50.     sendMsg:ok ?(v_i, e_i, o_i, TV_i) to msg.Sender ;

   procedure CheckOrder(o, TV)
51.  if o is stronger than o_i then  o_i ← o ; TV_i ← TV;
52.  remove nogoods and explanations incompatible with o_i;

   procedure CheckAgentView( )
53.  if ¬Consistent( v_i,AgentView ) then
54.     v_i ← ChooseValue( ) ;
55.     if (v_i) then sendMsg:ok ?(v_i, e_i, o_i, TV_i) to Succ(A_i);
56.     else Backtrack( );
57.  else if (o_i was modified) then
58.     sendMsg:ok ?(v_i, e_i, o_i, TV_i) to Succ(A_i) ;

   procedure Backtrack( )
59.  ng ← solve( NogoodStore ) ;
60.  if (ng = empty) then
61.     end ← true ; sendMsg:stp(system) ;
62.  ⟨x_k, o', TV', E'⟩ ←chooseVariableOrder( E_i, ng ) ;
63.  if TV' is smaller than TV_i then
64.     TV_i ← TV' ; o_i ← o' ; E_i ← E' ;
65.     setRhs( ng, x_k );
66.     sendMsg:ngd(ng, o_i, TV_i) to A_k ;
67.     remove e_k from E_i ;
68.     broadcastMsg:order(o_i, TV_i) ;
69.  else
70.     setRhs( ng, x_k );
71.     sendMsg:ngd(ng, o_i, TV_i) to A_k ;
72.  UpdateAgentView (x_k ← unknown) ;
73.  CheckAgentView( ) ;
```

FIGURE 1 – L'algorithme Agile-ABT ($1^{ère}$ Partie)

```
   procedure UpdateAgentView( Assignments )
74.  for each (var ∈ Assignments) do
75.     if (Assignments[var].t > AgentView[var].t ) then
76.        AgentView[var] ← Assignments[var];
77.  remove nogoods and explanations incoherent with AgentView;

   function ChooseValue( )
78.  for each v ∈ D_i not eliminated by NogoodStore do
79.     if ∃x_j ∈ AgentView such that ¬Consistent( v,x_j ) then
80.        add( x_j = v_j ⇒ x_i ≠ v, NogoodStore ) ;
81.  if D_i = ∅ then return( (empty) );
82.  else  t_i ← t_i+1 return( v );                    /* v ∈ D_i */
```

FIGURE 2 – L'algorithme Agile-ABT ($2^{ème}$ Partie)

variable et informe les agents moins prioritaires de sa décision (`CheckAgentView`, ligne 30) en envoyant des messages **ok?**.

Puis, une boucle considère la réception des différents types de messages. Si aucun message ne circule dans le réseau, l'état de quiescence est détecté par un algorithme spécialisé [5] et une solution globale est annoncée. La solution est l'ensemble des affectations courantes des variables.

Quand un agent A_i reçoit un message (de n'importe quel type), il vérifie si l'ordre inclus dans le message reçu est plus fort que son ordre courant o_i (`CheckOrder`, lignes 38, 42 et 44). Si c'est le cas, A_i remplace o_i et TV_i par ceux nouvellement reçus (ligne 51). Les nogoods et les explications qui ne sont plus compatibles avec o_i sont supprimés afin de garantir que $S(E_i)$ reste acyclique (ligne 52).

Si le message reçu est un message **ok?**, la Vue de A_i est mise à jour pour inclure les nouvelles affectations (`UpdateAgentView`, ligne 39). En plus de l'affectation de l'agent émetteur, A_i prend également les affectations contenues dans la partie gauche de l'explication transportée dans le message reçu afin de mettre à jour sa Vue. Ensuite, les nogoods et les explications qui ne sont plus cohérents avec la Vue de A_i sont supprimés (`UpdateAgentView` ligne 77). Après, si l'explication contenue dans le message reçu est valide, A_i met à jour l'ensemble des explications qu'il mémorise en ajoutant l'explication qu'il vient de recevoir. Enfin, la procédure `CheckAgentView` est appelée. (ligne 41).

A la réception d'un message **order**, A_i traite le nouvel ordre (`CheckOrder`) et la procédure `CheckAgentView` est appelée (ligne 43).

Quand A_i reçoit un message **ngd**, les procédures `CheckOrder` et `UpdateAgentView` (lignes 44 et 45) sont appelées. Ensuite, le nogood contenu dans le message est accepté s'il est cohérent avec la Vue de A_i et l'affectation de x_i et si ce nogood est compatible avec l'ordre courant de A_i. Sinon, le nogood est écarté et un message **ok?** est envoyé à l'émetteur comme dans ABT (lignes 49 et 50). Quand le nogood est accepté, il est mémorisé. Ce nogood justifiera la suppression de la valeur courante de A_i (ligne

47). Une nouvelle valeur qui soit consistante avec la Vue de A_i est recherchée (appel de `CheckAgentView`, ligne 48).

La procédure `CheckAgentView` vérifie si la valeur courante v_i est consistante avec la Vue de A_i. Si v_i est consistante, A_i vérifie si o_i a été modifié (ligne 57). Si c'est le cas, A_i doit envoyer son affectation aux agents moins prioritaires par le biais de messages **ok?**. Si v_i n'est pas consistante avec la Vue de A_i, ce dernier tente de trouver une valeur consistante (`ChooseValue`, ligne 54). Dans ce processus, quelques valeurs de A_i vont s'avérer inconsistantes. Dans ce cas, les nogoods justifiant la suppression de ces valeurs sont ajoutés au NogoodStore (`ChooseValue`, ligne 80). Si une nouvelle valeur consistante est trouvée, une explication e_i est construite et la nouvelle affectation est notifiée aux agents moins prioritaires que A_i par le biais de messages **ok?** (ligne 55). Sinon, chaque valeur de A_i est interdite par un nogood appartenant au NogoodStore et A_i doit faire un retour-arrière (`Backtrack`, ligne 56).

Dans la procédure `Backtrack`, A_i procède à la résolution de ses nogoods, et génère un nouveau nogood (ng). Si ng est vide, le problème n'a pas de solution. A_i termine l'exécution après l'envoi d'un message **stp** (ligne 61). Sinon, un des agents inclus dans ng doit changer sa valeur.

La fonction `chooseVariableOrder` sélectionne la variable x_k qui sera modifiée et un nouvel ordre o' tel que la nouvelle valeur de terminaison TV' est aussi petite que possible. Si TV' est inférieure à celle mémorisée par A_i, l'ordre courant et la valeur de terminaison sont remplacés par o' et TV' et A_i met à jour l'ensemble de ses explications par celui retourné par `chooseVariableOrder` (ligne 64). Ensuite, un message **ngd** est envoyé à l'agent A_k propriétaire de x_k (ligne 66). Puis, e_k est supprimée de E_i puisque A_k va probablement changer son explication après avoir reçu le nogood (ligne 67). Ensuite, A_i envoie un message **order** à tous les autres agents (ligne 68). Quand TV' n'est pas inférieure à la valeur de terminaison courante, A_i ne peut pas modifier l'ordre et la variable x_k qui sera modifiée est celle qui est la moins prioritaire selon l'ordre courant de A_i (lignes 70 et 71). Ensuite, l'affectation de x_k (la cible du backtracking) est supprimée de la Vue de A_i (ligne 72). Finalement, la recherche est continuée en faisant appel à la procédure `CheckAgentView` (ligne 73).

5 Preuves

Nous démontrons que Agile-ABT termine, qu'il est correct et complet. Nous démontrons également qu'il a une complexité spatiale polynomiale.

Théoreme 1 *Agile-ABT nécessite un espace $O(nd + n^2)$ par agent.*

Théoreme 2 *Agile-ABT est correct.*

Schéma de preuve. Quand l'état de quiescence est atteint, tous les agents connaissent nécessairement l'ordre o qui est l'ordre le plus fort calculé jusqu'à présent. En plus, tout agent A_i connait les affectations les plus récentes de tous ses prédécesseurs selon o. Par conséquent, toutes les contraintes entre A_i et ses prédécesseurs ont été testées avec succès par A_i. Sinon, A_i aurait tenté de changer son affectation et aurait envoyé un message ok? ou un message ngd, ce qui va casser l'état de quiescence. □

Théoreme 3 *Agile-ABT est complet.*

Preuve. Tous les nogoods sont générés par des inférences logiques à partir des contraintes existantes. Par conséquent, un nogood vide ne peut pas être déduit si une solution existe. □

La preuve de terminaison est construite en se basant sur les lemmes 1 et 2.

Lemme 1 *Pour chaque agent A_i, tant qu'une solution n'est pas trouvée et l'inconsistance du problème n'est pas prouvée, la valeur de terminaison mémorisée par A_i décroit après un temps fini.*

Schéma de preuve. Quand un agent est coincé suffisamment longtemps avec la même valeur de terminaison TV_i, tous les autres agents vont finir par avoir la même valeur de terminaison et le même ordre o auquel cette valeur de terminaison est associée. A partir de ce point de temps, Agile-ABT va fonctionner exactement comme ABT (ABT est un algorithme correct, il termine et il est complet [2]). Donc, soit une solution sera trouvée soit le premier agent dans l'ordre, $A_{o(1)}$, va recevoir un nogood avec une partie gauche lhs vide. Ce nogood va supprimer une des valeurs restantes du domaine de $A_{o(1)}$. Aussitôt que $A_{o(1)}$ envoie la nouvelle taille de son domaine courant (inférieure à tv^1) aux agents moins prioritaires dans o, tout agent qui fera un backtracking va générer une valeur de terminaison inférieure à TV_i. □

Lemme 2 *Soit $TV = [tv^1, \ldots, tv^n]$ une valeur de terminaison générée par un agent quelconque. Nous avons $tv^j \geq 0, \forall j \in 1..n$*

Schéma de preuve. Toues les explications e_k mémorisées par un agent A_i ont $rhs(e_k) \geq 1$ puisque $rhs(e_k)$ représente la taille du domaine courant de A_k. Les valeurs de terminaison sont construites avec les rhs des explications. L'unique cas où tv^j peut être égal à 0 (ligne 8) est lorsque $A_{o(j)}$ est sélectionné par A_i comme cible du backtracking, et dans un tel cas l'explication $e_{o(j)}$ est supprimée juste après l'envoi du message nogood à $A_{o(j)}$ (ligne 67). Donc, A_i ne va, à aucun moment, mémoriser une explication e_k avec $rhs(e_k) = 0$ et donc il ne pourra jamais produire une valeur de terminaison avec des valeurs négatives. □

Théoreme 4 *Agile-ABT termine.*

Preuve. Preuve directe à partir des lemmes 1 et 2. □

6 Évaluation expérimentale

Nous avons comparé Agile-ABT à ABT, ABTDO et ABTDO avec les heuristiques rétroactives. Toutes nos expériences ont été réalisées sur la plateforme DisChoco [3] où les agents sont simulés par des threads java qui communiquent en se passant des messages. Nous évaluons la performance des algorithmes par le coût de communication et l'effort de calcul. Le coût de communication est mesuré par le nombre total de messages échangés entre les agents durant l'exécution de l'algorithme ($\#msg$). Ce nombre total inclut les messages échangés pour détecter la terminaison (messages système). L'effort de calcul est mesuré par une adaptation du nombre de tests non-concurrents de contraintes ($\#nccc$) (non-concurrent constraint checks) qui prend en considération les tests de nogoods.

Pour ABT, nous avons implémenté la version standard. Pour ABTDO [11], nous avons implémenté la meilleure version basée sur l'heuristique *nogood-triggered*. Dans cette version le récepteur d'un nogood modifie l'ordre de telle sorte que l'émetteur du nogood soit placé immédiatement après le récepteur.

Pour ABTDO avec heuristiques rétroactives [12], nous avons implémenté la meilleure version où un agent qui génère un nogood après avoir vidé son domaine, tente de modifier l'ordre en essayant de se placer entre le dernier et l'avant dernier agent dans le nogood. Cependant, le générateur du nogood ne peut se placer avant un autre agent A_k que si la taille du domaine courant du générateur du nogood est inférieure à la taille du domaine courant de A_k (noté par ABTDO-retro).

Les DisCSP binaires uniformes aléatoires

Ces algorithmes sont testés sur des DisCSP uniformes binaires $\langle n,\, d,\, p_1,\, p_2\rangle$, avec n agents/variables, d valeurs par variable, une connectivité définie comme étant la proportion p_1 des contraintes binaires existantes, et une dureté de contraintes définie comme étant la proportion p_2 des paires de valeurs interdites. Nous présentons les résultats pour deux classes de problèmes : des problèmes avec une faible densité $\langle 20, 10, 0.20, 0.66\rangle$ et des problèmes avec une densité élevée $\langle 20, 10, 0.70, 0.30\rangle$. Les deux classes sont dans le pic de complexité. Pour chaque classe, nous avons généré 25 instances, chaque instance a été résolue 4 fois. Nous reportons les moyennes de 100 exécutions.

La table 1 présente les résultats sur les problèmes aléatoires. En terme d'effort de calcul ($\#nccc$), ABT est l'algorithme le moins performant. ABTDO-ng est largement meilleur que ABT et ABTDO-retro est plus performant que ABTDO-ng. Ces constats sont similaires à ceux qui ont été formulés dans [12]. Agile-ABT domine tous ces algorithmes. Ces résultats suggèrent que dans le pic de complexité, plus l'algorithme est sophistiqué, plus il est performant. En ce qui concerne le coût de communication ($\#msg$), les messages supplémentaires envoyés par les algorithmes de changement d'ordre n'impliquent pas nécessairement une résolution plus rapide des problèmes. ABT nécessite moins de messages que ABTDO-ng pour les deux classes de problèmes, et moins de messages que ABTDO-retro sur les problèmes denses. Par contre, Agile-ABT est l'algorithme qui nécessite le moins de messages, même quand il est comparé à ABT. Ce n'est pas uniquement parce que Agile-ABT termine plus rapidement que les autres algorithmes, c'est aussi parce que Agile-ABT est plus parcimonieux dans le choix des nouveaux ordres que les autres algorithmes. Les valeurs de terminaison semblent limiter les changements d'ordres à ceux qui seront plus rentables.

TABLE 1 – Problèmes aléatoires

Algorithme	faible densité		densité élevée	
	$\#nccc$	$\#msg$	$\#nccc$	$\#msg$
ABT	710,911	213,702	14,015,644	3,947,616
ABTDO-ng	246,357	336,693	20,545,772	18,546,859
ABTDO-retro	80,908	136,146	8,844,655	8,727,913
Agile-ABT	46,287	55,121	2,432,920	1,321,120

Les problèmes distribués de capteur-mobile

Le problème distribué de capteur-mobile [1] est un benchmark basé sur un problème distribué réel. Il consiste en m capteurs qui traquent n mobiles. Chaque mobile doit être traqué par 3 capteurs. Chaque capteur peut traquer au plus un mobile. Une solution doit satisfaire les contraintes de visibilité et de compatibilité. Les contraintes de visibilité définissent l'ensemble des capteurs qui sont visibles pour chaque mobile. Les contraintes de compatibilité définissent les capteurs qui sont compatibles entre eux. Les problèmes sont caractérisés par $\langle m,\, n,\, p_c,\, p_v\rangle$ avec p_c est la probabilité que deux capteurs soient compatibles et p_v est la probabilité qu'un capteur soit visible à un mobile. Nous présentons les résultats pour deux classes de problèmes : des problèmes sous-contraints $\langle 25, 5, 0.6, 0.4\rangle$ et des problèmes dans le pic de complexité $\langle 25, 5, 0.6, 0.6\rangle$. Nous avons généré 25 instances et chaque instance a été résolue 4 fois. La moyenne est calculée pour 100 exécutions.

Le tableau 2 présente les résultats. La classe $\langle 25, 5, 0.6, 0.4\rangle$ est intéressante parce que dans cette région, les procédures de recherche font beaucoup d'aller-retours lorsque de mauvaises décisions sont prises au début de la recherche. C'est vraisemblablement la raison pour laquelle ABT est l'algorithme le moins performant en terme de $\#nccc$. ABTDO-retro est le meilleur algorithme sur ces problèmes. Au pic, tous les algorithmes ont une

TABLE 2 – Capteur-mobile

Algorithme	$p_c = 0.6, p_v = 0.4$		$p_c = 0.6, p_v = 0.6$	
	$\#nccc$	$\#msg$	$\#nccc$	$\#msg$
ABT	1,725,258	84,099	742,998	51,477
ABTDO-ng	312,400	111,511	845,475	350,930
ABTDO-retro	63,784	30,429	832,141	417,526
Agile-ABT	154,734	7,103	331,994	39,172

performance similaire, à l'exception de Agile-ABT qui est deux fois plus rapide. En ce qui concerne les $\#msg$, ABTDO-retro domine ABT sur $\langle 25, 5, 0.6, 0.4\rangle$ parce qu'il termine plus rapidement que ABT. En dehors de cela, les choses sont semblables aux problèmes aléatoires, où les algorithmes ABTDO et ABTDO-retro sont pénalisées par les messages de changements d'ordre alors que le coût de communication de Agile-ABT reste faible.

7 Conclusion

Nous avons proposé Agile-ABT, un algorithme qui est capable de modifier l'ordre d'une manière plus agile que les approches précédentes. Grâce au concept des valeurs de terminaison, Agile-ABT est capable de choisir une cible du backtracking qui n'est pas nécessairement l'agent ayant la plus faible priorité parmi les agents conflictuels. En plus, l'ordre des agents qui apparaissent avant la cible du backtracking peut être modifié. Ces caractéristiques intéressantes sont inhabituelles pour un algorithme de complexité spatiale polynomiale. Nos expériences confirment l'importance de ces caractéristiques.

Références

[1] R. Bejar, C. Domshlak, C. Fernandez, K. Gomes, B. Krishnamachari, B.Selman et M.Valls. Sensor networks and distributed CSP : communication, computation and complexity. *Artificial Intelligence*, 161 :1-2:117–148, 2005.

[2] C. Bessiere, A. Maestre, I. Brito et P. Meseguer. Asynchronous backtracking without adding links : a new member in the ABT family. *Artificial Intelligence*, 161 :1-2:7–24, 2005.

[3] R. Ezzahir, C. Bessiere, M. Belaissaoui et E. H. Bouyakhf. Dischoco : a platform for distributed constraint programming. Dans *Proceeding of Workshop on DCR of IJCAI-07*, pages 16–21, 2007.

[4] Matthew L. Ginsberg et David A. McAllester. Gsat and dynamic backtracking. Dans *KR*, pages 226–237, 1994.

[5] K. Chandy Mani et L. Lamport. Distributed snapshots : determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, 1985. ISSN 0734-2071.

[6] A. Petcu et B. Faltings. A value ordering heuristic for distributed resource allocation. Dans *Proceeding of CSCLP04*, Lausanne, Switzerland, 2004.

[7] M. C. Silaghi. Generalized dynamic ordering for asynchronous backtracking on DisCSPs. Dans *DCR workshop, AAMAS-06*, Hakodate, Japan, 2006.

[8] M.-C. Silaghi, D. Sam-Haroud et B. Faltings. Hybridizing ABT and AWC into a polynomial space, complete protocol with reordering. Rapport technique, 2001.

[9] M. Yokoo. Asynchronous weak-commitment search for solving distributed constraint satisfaction problems. Dans *Proceeding of CP*, pages 88–102, Cassis, France, 1995.

[10] M. Yokoo, E. H. Durfee, T. Ishida et K. Kuwabara. Distributed constraint satisfaction problem : Formalization and algorithms. *IEEE Trans. on Data and Kn. Eng.*, 10:673–685, 1998.

[11] R. Zivan et A. Meisels. Dynamic ordering for asynchronous backtracking on discsps. *Constraints*, 11 (2-3):179–197, 2006. ISSN 1383-7133.

[12] R. Zivan, M. Zazone et A.Meisels. Min-domain retroactive ordering for asynchronous backtracking. *Constraints*, 14(2):177–198, 2009. ISSN 1383-7133.

Contrôle statistique du processus de propagation de contraintes

Frédéric Boussemart Fred Hemery Christophe Lecoutre Mouny Samy Modeliar

CRIL - CNRS UMR 8188, Université Lille Nord de France, Artois, France
{boussemart,hemery,lecoutre,modeliar}@cril.fr

Résumé

Dans cet article, nous nous intéressons à la propagation de contraintes, un processus de filtrage qui est typiquement exécuté à chaque étape d'un algorithme de recherche en profondeur d'abord avec retours-arrière tel que MAC. A partir d'une analyse statistique portant sur des propriétés précises du mécanisme de propagation, nous montrons qu'il est possible d'effectuer des prédictions fiables sur la capacité du processus de propagation à détecter l'incohérence. En utilisant cette observation dans le but de contrôler l'effort de propagation, nous montrons son intérêt pratique sur un jeu d'essai comportant de nombreuses séries d'instances CSP.

Abstract

In this paper, we investigate constraint propagation, a mechanism that is run at each basic step of a backtrack search algorithm such as the popular MAC. From a statistical analysis of some relevant features concerning propagation on a large set of CSP instances, we show that it is possible to make reasonable predictions about the capability of constraint propagation to detect inconsistency. Using this observation in order to control propagation effort, we show its practical effectiveness.

1 Introduction

De nombreux problèmes de décision sont de nature combinatoire, et peuvent être modélisés à l'aide de variables discrètes combinées par des contraintes. On obtient alors des instances du problème de satisfaction de contraintes (CSP pour Constraint Satisfaction Problem) qui sont représentés formellement par des réseaux de contraintes (CN pour Constraint Network) [9, 12]. Résoudre une instance CSP (ou résoudre un CN) consiste (dans cet article) à trouver une solution ou prouver qu'aucune solution n'existe.

La recherche avec retours-arrière (backtrack search) est utilisée couramment pour résoudre les instances CSP. Pour ce type de recherche, une exploration en profondeur d'abord permet d'instantier les variables et un mécanisme de retours-arrière permet de gérer les impasses détectées. Après chaque instantiation de variable, un processus de filtrage est exécuté sur la base d'une propriété locale, appelée cohérence (consistency), qui permet d'élaguer certaines parties de l'espace de recherche ne contenant aucune solution. La cohérence d'arc généralisée (GAC pour Generalized Arc Consistency) est la forme de cohérence locale la plus forte lorsque les contraintes sont considérées de manière indépendante. Intuitivement, GAC permet d'éliminer en toute sécurité une valeur a du domaine d'une variable x si il existe une contrainte c impliquant x mais n'acceptant aucun tuple (construit sur la base des domaines courants des variables) avec la valeur a pour x. Aussitôt qu'une inférence locale (par exemple, correspondant à l'élimination d'une valeur) est effectuée, les conditions pour déclencher de nouvelles inférences peuvent potentiellement être vérifiées puisque les variables sont typiquement partagées par plusieurs contraintes. Cette manière de propager le résultat d'inférences locales de contraintes en contraintes est appelée *propagation de contraintes* (par exemple, voir [2]).

L'un des algorithmes de recherche avec retours-arrière les plus utilisés est appelé MAC (Maintaining Arc Consistency) [21]. Pendant la recherche, MAC maintient en permanence GAC de manière à réduire la taille de l'arbre de recherche à explorer. GAC permet d'éliminer de nombreuses valeurs inutiles, mais malheureusement établir cette propriété constamment en cours de recherche peut s'avérer très coûteux. FC [11] est un autre algorithme de recherche avec retours-arrière qui réalise une forme très limitée de propagation de contraintes : seules les contraintes impliquant la dernière variable assignée sont "propagées". Bien que MAC soit considérée comme état de l'art, sur certains problèmes, FC peut parfois surclasser MAC.

Plusieurs tentatives pour ajuster le bon niveau de filtrage ont été effectuées ces dernières années. Tout d'abord, dans [8], les auteurs proposent une vison paramétrée de la cohérence locale qui permet de nombreuses formes partielles de GAC d'être établies. Le paramètre utilisé pour cette approche est la distance entre la dernière variable assignée et les autres variables dans le graphe de contraintes. Ensuite, Mehta et van Dongen ont introduit une approche probabiliste [20] de manière à n'effectuer uniquement que les tests de contraintes utiles en ne recherchant pas de support pour une valeur lorsqu'il y a une forte probabilité qu'un tel support existe. Cette technique d'inférence de support probabiliste a été étudiée à la fois pour GAC et SAC (singleton arc consistency). Plus récemment, Stergiou [23] a proposé d'adapter dynamiquement le niveau de cohérence locale devant être appliquée durant la recherche. Il a montré que l'information concernant les domaines devenus vides (domain wipe-outs) et les suppressions de valeurs pendant la recherche peuvent être utilisées non seulement pour sélectionner les variables mais également pour ajuster automatiquement le niveau de filtrage

Dans cet article, à partir d'une analyse de nature statistique concernant la propagation de contraintes, nous montrons qu'il est possible de faire des prédictions raisonnables sur la capacité de la propagation à détecter l'incohérence. Par une corrélation existant entre deux critères importants de la propagation (établie à partir d'un large jeu d'essai), à savoir la longueur et le résultat de la propagation, nous montrons qu'il est possible de limiter le coût de la propagation de contraintes de manière originale. Cette propriété se révèle prometteuse, illustrée ici avec une variante de MAC.

2 Préliminaires

Un réseau de contraintes discret (CN) P est composé d'un ensemble fini de variables, noté $vars(P)$, et d'un ensemble fini de contraintes, noté $cons(P)$. Chaque variable x possède un domaine, noté $dom(x)$, qui contient l'ensemble fini des valeurs qui peuvent être assignées à x. Chaque contrainte c porte sur un ensemble ordonné de variables, appelé portée et noté $scp(c)$. L'ensemble des tuples autorisés pour les variables impliquées dans c est défini formellement par une relation notée $rel(c)$, qui peut être donnée en extension ou en intention. L'arité d'une contrainte c est la taille de $scp(c)$. Une contrainte binaire porte sur exactement 2 variables, et une contrainte non-binaire sur plus de 2 variables.

Les CN sont utiles pour représenter de nombreux problèmes combinatoires comme, par exemple, le problème de coloration de graphe : étant donné un graphe $G = (V, E)$ et k couleurs, est-il possible d'associer une couleur à chaque sommet tel que les deux sommets de chaque arête soient coloriés différemment ? Ceci peut être modélisé par un CN P tel que (i) pour chaque sommet $v \in V$, il existe une variable x_v dans $vars(P)$ avec $dom(x) = \{1, 2, \ldots, k\}$, et (ii) pour chaque arête $e = \{v, v'\} \in E$, il existe une contrainte c_e dans $cons(P)$ tel que $scp(c_e) = \{x_v, x_{v'}\}$ et sa sémantique est donnée par $x_v \neq x_{v'}$.

Une solution à un CN est l'assignation d'une valeur à chaque variable telle que toutes les contraintes soient satisfaites. Un CN est dit *cohérent* ssi il admet au moins une solution. Le problème de satisfaction de contraintes (CSP) consiste à déterminer si un CN donné est cohérent ou non. Aussi, une instance CSP est définie par un CN qui est résolu en trouvant une solution ou en prouvant qu'aucune solution n'existe. Dans de nombreux cas, une instance CSP peut être résolue par des techniques de recherche et d'inférence [9, 12].

Une cohérence locale est une propriété (ou condition) générale définie sur les CN. Typiquement, la non vérification d'une cohérence locale nous permet de faire des déductions, également appelées inférences, qui révèlent des nogoods (standards), i.e. révèlent des instantiations (partielles) de variables qui ne peuvent mener à des solutions. La cohérence d'arc généralisée (GAC) est la propriété centrale en programmation par contraintes.

Étant donné un ensemble ordonné $\{x_1, \ldots, x_i, \ldots, x_r\}$ de r variables et un r-tuple $\tau = (a_1, \ldots, a_i, \ldots, a_r)$ de valeurs, la valeur a_i sera notée $\tau[x_i]$. Un r-tuple τ est *valide* sur une contrainte c d'arité r ssi $\forall x \in scp(c)$, $\tau[x] \in dom(x)$. Rappelons qu'un tuple τ est *autorisé*, ou accepté, par une contrainte c ssi $\tau \in rel(c)$. Un r-tuple τ est un *support* sur une contrainte c d'arité r ssi τ est un tuple valide sur c qui est également autorisé par c. Si τ est un support sur une contrainte c impliquant une variable x et tel que $\tau[x] = a$, on dit alors que τ est un support pour (x, a) sur c ; on dit également que (x, a) est supporté par c.

Une v-valeur d'un CN P est un couple variable-valeur (x, a) tel que $x \in vars(P)$ et $a \in dom(x)$. Une v-valeur (x, a) d'un CN P est *GAC-cohérente* sur P ssi pour toute contrainte c de P impliquant x, il existe un support pour (x, a) sur c. Une contrainte c est *GAC-cohérente* ssi $\forall x \in scp(c), \forall a \in dom(x)$, il existe un support pour (x, a) sur c. Un CN P est *GAC-cohérent* ssi chaque contrainte de P est GAC-cohérente. Si une v-valeur (x, a) n'est pas GAC-cohérente, elle est dite *GAC-incohérente*. Il est aisé de constater que toute v-valeur GAC-incohérente ne peut apparaître dans aucune solution et, par conséquent, est (globalement) incohérente. Rétablir GAC signifie rendre le CN GAC-cohérent en éliminant toutes les v-valeurs GAC-incohérentes. Le résultat de l'établissement de GAC est un CN unique, noté $GAC(P)$, qui est appelé la GAC-*fermeture* de P ; voir par exemple, [2]. De nombreux algorithmes ont été proposés pour établir (G)AC. Comme exemples d'algorithmes génériques, citons (G)AC3 [17, 18], (G)AC2001 [4] et GAC3^{rm} [14]. Pour les contraintes exprimées en extension, on trouve des développements récents [16, 15, 10, 7, 13].

3 Propagation orientée variable

Pour établir une cohérence donnée sur un réseau de contraintes, des déductions locales (inférences) sont effectuées de manière itérative jusqu'à l'obtention d'un point fixe. Étant donné que les variables sont typiquement impliquées dans plusieurs contraintes, toute inférence effectuée peut en induire de nouvelles. Le mécanisme qui consiste à propager le résultat d'inférences locales de contraintes en contraintes est appelée propagation de contraintes, et est mis en oeuvre par des algorithmes de filtrage.

Classiquement, seuls les événements concernant les variables sont utilisés pour la propagation de contraintes. Dans un contexte de filtrage générique (à gros grain), qui exécute une même procédure indépendamment de la nature des contraintes, la modification du domaine d'une variable (suppression d'une ou plusieurs valeurs) est le seul type d'événement retenu. Le filtrage à grain fin est quant à lui guidé par les valeurs qui sont supprimées.

Dans cette section, nous donnons une description d'un schéma de propagation générique à gros grain qui peut être utilisé pour appliquer GAC sur un réseau de contraintes donné. Les algorithmes de propagation à gros grain effectuent des révisions successives d'arc. Un arc est représenté par un couple (c, x) où c est une contrainte et x une variable contenue dans $scp(c)$. La révision d'un arc (c, x) supprime de $dom(x)$ chaque valeur qui n'a pas de support sur c. Une révision est dite $effective$ si elle permet de supprimer au moins une valeur. Une révision est dite $inutile$ si on peut prédire qu'elle sera infructueuse. De manière évidente, il faut éviter d'effectuer des révisions inutiles.

Dans le *schéma orienté variable* [19, 24, 5], lorsqu'une valeur du domaine d'une variable est supprimée, la variable est ajoutée à un ensemble Q appelé *queue de propagation*. Le *schéma orienté variable* peut être optimisé par l'utilisation de tampons temporels (timestamps). Ceux-ci permettent de dater certains événements et de suivre au cours du temps la progression d'un algorithme. Ils permettent notamment dans notre cas de déterminer si une révision est inutile ou pas.

En introduisant une horloge (compteur) globale $time$ et en associant un tampon $stamp[x]$ à chaque variable x et un tampon $stamp[c]$ à chaque contrainte c, il est possible de déterminer les révisions qui seront effectives. La valeur $stamp[x]$ indique à quel moment a eu lieu la dernière suppression d'une valeur dans $dom(x)$ tandis que la valeur $stamp[c]$ indique le moment le plus récent où la contrainte c a été rendue GAC-cohérente. Les variables $time$, $stamp[x]$ pour chaque variable x et $stamp[c]$ pour chaque contrainte c sont initialisées à 0. La valeur de $time$ est incrémentée à chaque fois qu'une variable est ajoutée à Q et à chaque fois qu'une contrainte est rendue GAC-cohérente.

L'appel $\mathsf{GAC}^{var}(vars(P))$, voir l'algorithme 1, assure la cohérence d'arc généralisée pour un réseau de contraintes P donné ; dans un premier temps nous passons sous silence les instructions en gris clair des lignes 4,7 et 17-18. La valeur booléenne $false$ est retournée si P est prouvé GAC-incohérente, c'est-à-dire si $GAC(P) = \bot$. Soit $past(P)$ l'ensemble des variables passées de P, c'est-à-dire les variables de P qui ont été instanciées par un algorithme de recherche avec retours-arrière comme FC ou

Algorithme 1: $\mathsf{GAC}^{var}(X_{evt}$: ensemble de variables)

Résultat : $true$ iff $GAC(P) \neq \bot$

```
1  Q ← ∅
2  foreach variable x ∈ X_evt do
3      insert(Q, x)
4  cnt ← 0
5  while Q ≠ ∅ do
6      choisir et supprimer x de Q
7      cnt ← cnt + 1
8      foreach
       c ∈ cons(P) | x ∈ scp(c) ∧ stamp[x] > stamp[c] do
9          foreach variable y ∈ scp(c) | y ∉ past(P) do
10             if y ≠ x or ∃z ∈ scp(c) | z ≠
               x ∧ stamp[z] > stamp[c] then
11                 if revise(c, y) then
12                     if dom(y) = ∅ then
13                         return false
14                     insert(Q, y)
15         time ← time + 1
16         stamp[c] ← time
17     if cnt ≥ threshold then
18         break
19 return true
```

Algorithme 2: insert(Q : ensemble de variables, x : variable)

```
1  Q ← Q ∪ {x}
2  time ← time + 1
3  stamp[x] ← time
```

Algorithme 3: revise(c : contrainte, x : variable)

Résultat : $true$ ssi la révision de (c, x) est effective

```
1  nbElements ← |dom(x)|
2  foreach value a ∈ dom(x) do
3      if ¬seekSupport(c, x, a) then
4          supprimer a de dom(x)
5  return nbElements ≠ |dom(x)|
```

MAC, qui sera décrit dans la suite. Lors de l'initialisation, nous avons $past(P) = \emptyset$ car il n'y a aucune variable assignée et GAC$^{arc}(vars(P))$ calcule simplement la GAC-fermeture de P.

Pour établir GAC, les variables sont sélectionnées de manière itérative dans Q (ligne 6). Chaque contrainte c qui porte sur la variable sélectionnée x est prise en compte si $stamp[x] > stamp[c]$. Dans ce cas, chaque variable non assignée $y \neq x$ dans $scp[c]$ est révisée par rapport à la contrainte c. Chaque révision est effectuée par la fonction revise, de l'algorithme 3, qui retourne $true$ si une valeur, au moins, a été supprimée. Si la révision est effective, y est ajoutée à Q. Pour garantir que c est encore GAC-cohérente, nous devons déterminer si l'un des domaines des autres variables que x dans c a été modifié depuis le dernier établissement de GAC sur c. Ceci est pris en charge par la seconde partie de la condition de la ligne 10 de l'algorithme 1. Si la seconde partie de la condition est vérifiée alors x est une variable qui doit être révisée. Si un domaine devient vide, l'algorithme 1 retourne $false$ à la ligne 13. Dans l'algorithme 3, pour chaque valeur a dans $dom(x)$ la fonction seekSupport indique si il existe un support pour (x, a) sur c. De nombreuses implémentations de seekSupport ont été proposées telles que celles utilisées avec (G)AC3, GAC2001 and GAC3^{rm}.

4 Contrôle de la propagation

Dans cet article, nous considérons l'algorithme MAC (Maintaining Arc Consistency), un algorithme de recherche complète considéré comme l'un des plus efficaces pour la résolution générique d'instances CSP. MAC [21] parcourt l'espace de recherche en profondeur d'abord avec retours-arrière et établit la propriété d'arc cohérence (généralisée) après chaque décision prise en cours de recherche. Il construit un arbre de recherche binaire tel que, à chaque noeud interne v, un couple (x, a) est sélectionné, x étant une variable non assignée, et a une valeur appartenant à $dom(x)$. On considère alors deux cas : l'assignation $x = a$ (décision positive) et la réfutation $x \neq a$ (décision négative).

Une variable *passée* est (explicitement) assignée, tandis qu'une variable *future* n'est pas (explicitement) assignée. Pour maintenir GAC au cours de la recherche, on fait appel à GAC$^{var}(\{x\})$ après chaque décision (positive ou négative) prise sur une variable x. Dans le cas d'une décision positive, nous savons que x appartient désormais aux variables passées. En conséquence, la queue Q ne contient initialement que la variable x.

Par la suite, nous nous référons également à *forward checking* (FC), algorithme de recherche [19, 11] qui maintient une forme partielle d'arc cohérence (généralisée). Au niveau des réseaux de contrainte binaires, lorsqu'une variable x est assignée, seules les variables futures (non assignées) directement connectées à x sont révisées. Le filtrage via FC n'est pas exécuté en phase de pré-traitement (i.e. avant la recherche, lorsqu'aucune variable n'est assignée), ni après une décision négative. Remarquons que, dans le cadre de réseaux de contraintes non binaires, il existe différentes possibilités de généralisation de FC [3].

Nous nous intéressons dans cet article à deux propriétés particulières de la propagation :

- la *longueur* de la propagation, qui correspond au nombre de variables qui ont été extraites de la queue via un appel à la fonction GAC^{var},
- et le *résultat* de la propagation qui correspond à la valeur, $true$ ou $false$, renvoyée par GAC^{var}.

Plus précisément, nous avons étudié l'existence d'une corrélation entre la longueur de la propagation, et son résultat. On observe alors que pour de nombreuses séries d'instances [1], la longueur moyenne de la propagation est significativement plus petite lorsque la fonction GAC^{var} renvoie $false$ que lorsqu'elle renvoie $true$.

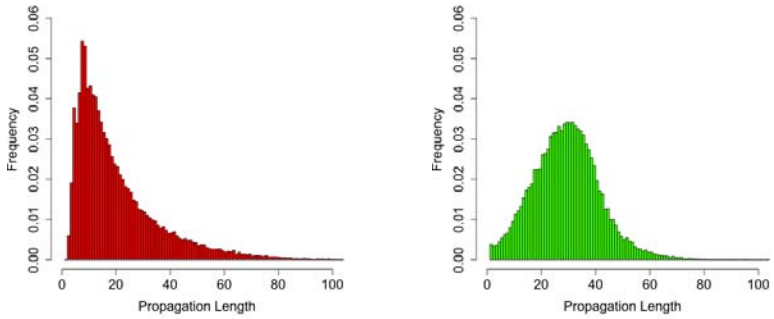
Par exemple, lors de la résolution de l'instance de mots croisés cw-words-vg5-9, MAC (avec l'heuristique *dom/wdeg*) appelle environ 91 000 fois GAC^{var}. 58 000 appels renvoient $false$ et 33 000 renvoient $true$. La distribution précise (en pourcentages) est donnée par la figure 1(a). La longueur moyenne des résultats $false$ (notée avg-false) est de 20.8, tandis qu'elle est de 29.7 pour les résultats $true$ (avg-true). D'autres cas de figure présentent des écarts encore plus prononcés, comme le montrent très clairement les figures 1 et 2. Les valeurs avg-false et avg-true suivantes sont obtenues :

- 4.4 et 24.7 pour l'instance val8-13
- 8.8 et 28.3 pour l'instance langford-3-12
- 9.8 et 12.8 pour l'instance qcp-15-120-5
- 4.2 et 12.6 pour l'instance rand-3-20-20-60-632-18
- 2.7 et 23.9 pour l'instance scens11-f4
- 2.0 et 45.3 pour l'instance lemma-50-9-mod

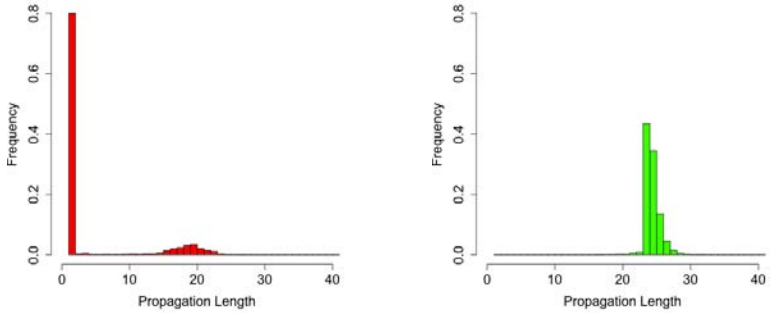
Précisons que chaque résultat présenté ci-dessus est représentatif du fonctionnement général de MAC sur l'ensemble de la série à laquelle appartient l'instance. La valeur de avg-false est en général bien plus petite que celle de avg-true, sauf dans quelques cas, assez rares, où l'inverse est constaté. Par exemple, au niveau de la série pseudo-ii, on obtient de manière assez surprenante 58.9 pour avg-false, et 1.1 pour avg-true sur l'instance pseudo-ii32c1 (voir figure 1(c)).

La mise en évidence d'une corrélation entre la longueur et le résultat de la propagation nous a amenés à développer une nouvelle variante de MAC. Globalement, si nous pouvons déterminer une longueur de propagation à partir de laquelle la fonction GAC^{var} a de fortes probabilités de retourner $true$, pourquoi ne pas l'arrêter dès que cette lon-

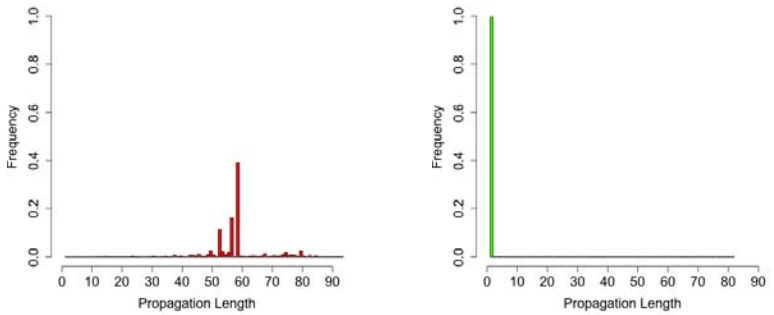
1. Nos expérimentations ont porté sur un nombre conséquent de séries disponibles à `http://www.cril.fr/~lecoutre/benchmarks.html`.



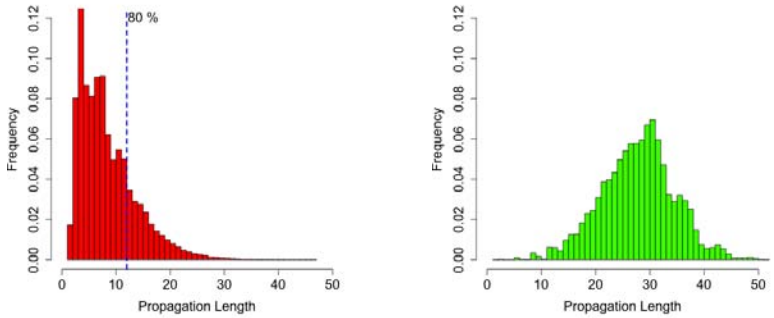
(a) crossword-m1c-words-vg5-9



(b) graph-valiente-8-13



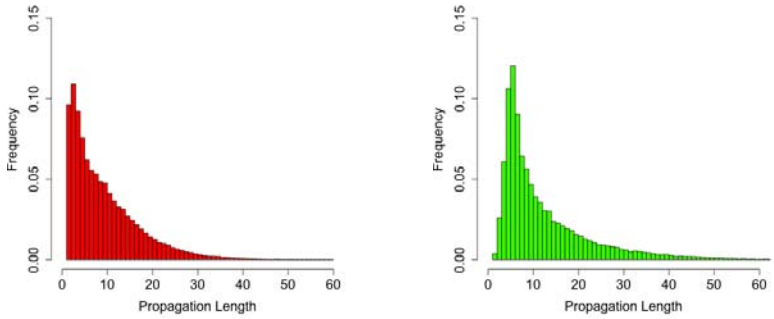
(c) pseudo-ii32c1



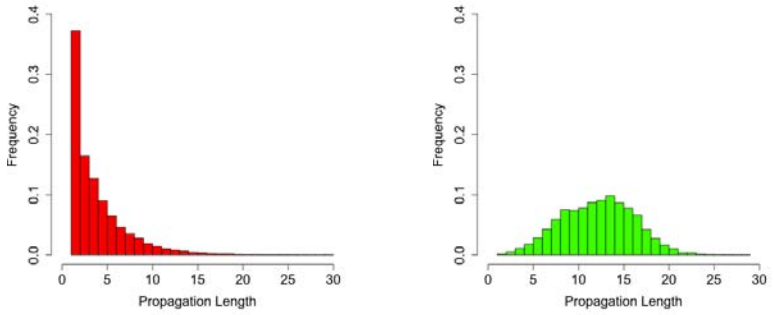
(d) langford-3-12

FIGURE 1 – Corrélation entre longueurs et résultats de propagation : distribution des résultats de propagation $false$ (à gauche) et $true$ (à droite) pour différentes instances.

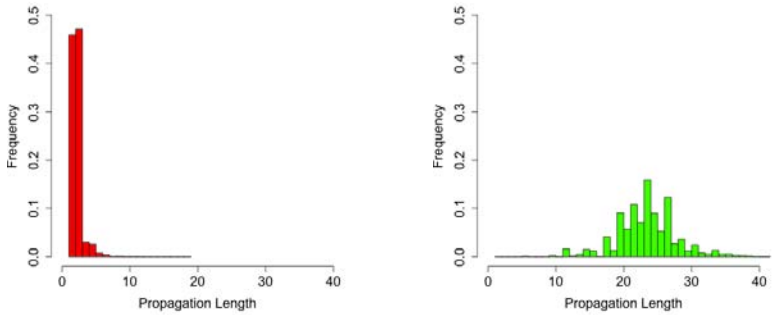
r



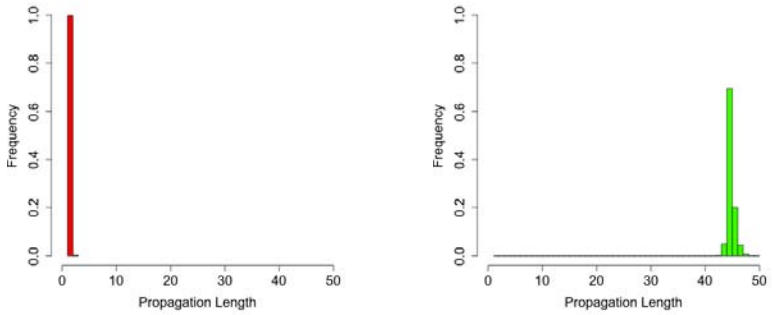
(a) qcp-15-120-5



(b) rand-3-20-20-60-632-18



(c) scens11-f4



(d) lemma-50-9-mod

FIGURE 2 – Corrélation entre longueurs et résultats de propagation : distribution des résultats de propagation $false$ (à gauche) et $true$ (à droite) pour différentes instances.

gueur est atteinte ? Certaines valeurs qui auraient dû être supprimées seront conservées, mais en contrepartie, nous diminuons significativement l'effort nécessaire pour identifier les situations d'échec engendrées par un domaine vide. Nous proposons pour cela MAC^c, qui correspond à l'algorithme 1 dont les lignes grises additionnelles, entraînent les modifications suivantes :

- un compteur cnt est utilisé : initialisé à zéro (ligne 4), il est incrémenté à chaque fois qu'une variable est prise dans Q (ligne 7) ;
- une variable entière $threshold$ est ajoutée : elle correspond à la longueur à laquelle la propagation doit être stoppée ;
- un nouveau test apparaît en fin de boucle (lignes 17-18) pour stopper prématurément la propagation lorsque cela paraît utile.

Remarquons que si $threshold$ vaut 1, MAC^c est équivalent à FC, tandis que si $threshold$ est positionné à l'infini, MAC^c devient équivalent à MAC. En d'autres termes, MAC^c va en général effectuer plus de propagation que FC, mais moins que MAC.

Il nous faut maintenant trouver une bonne valeur de seuil ($threshold$) lors de la résolution d'une instance. A partir de distributions telles que celles des figures 1 et 2, on peut en premier lieu calculer, pour chaque longueur n, le coût moyen $AC(n)$ de détection d'un domaine vide (résultat $false$) lorsque le seuil vaut n. Pour une longueur n, le coût moyen $AC(n)$ est calculé comme suit :

$$\frac{\sum_{i=1}^{n} i \times (F_i + T_i) + \sum_{i=n+1}^{\infty} n \times (F_i + T_i)}{\sum_{i=1}^{n} F_i}$$

où T_i et F_i correspondent respectivement aux nombres de résultats $true$ et $false$ lorsque la longueur est égale à i. Ce coût est exprimé en nombre d'opérations de base effectuées par GAC^{var}, c'est-à-dire en nombre de fois où une variable est extraite de Q. Cette formule signifie que, pour un entier i compris entre 1 et n, le coût est égal à i multiplié par le nombre de fois où la longueur a été égale à i, et pour tout entier i plus grand que n, le coût est égal à n (car la propagation a été stoppée à ce niveau) multiplié par le nombre de fois où la longueur a été égale à i. Ce coût global est alors divisé par le nombre de fois où GAC^{var} a renvoyé $false$ lorsque la longueur était égale à i. Une illustration est donnée par les figures 3, 4 et 5.

Nous pouvons alors contrôler la propagation en positionnant le seuil à la valeur de i qui détermine le coût le plus bas. Dans l'hypothèse où nous connaîtrions à l'avance la distribution, nous obtiendrions, par exemple, des seuils égaux à 1 pour l'instance val8-13, à 12 pour langford-3-12, ou encore à 149 pour pseudo-ii32c1 (voir les figures 1(b), 1(d) et 1(c)). En d'autres termes, MAC^c pourrait identifier 80% des choix conduisant à un domaine vide pour les instances val8-13 et langford-3-12, et 100% pour pseudo-ii32c1 (voir les figures 1(b), 1(d) et 1(c)). Cela signifie également que si de telles valeurs de seuil avaient été fixées dès le début de la résolution, MAC^c se serait comporté de manière similaire à FC sur l'instance val8-13, et de manière similaire à MAC sur l'instance pseudo-ii32c1. En pratique, il n'est pas possible de déterminer ces valeurs à l'avance, aussi nous proposons une procédure dynamique et adaptative de calcul du seuil au cours de la recherche, ce qui produit des variations des valeurs du seuil autour des valeurs théoriques présentées plus haut. Sur nos exemples, MAC^c se comporte alors de façon légèrement différente de FC ou de MAC.

Au niveau de notre implémentation, nous avons utilisé le coût moyen minimum comme valeur de seuil. En fait, le

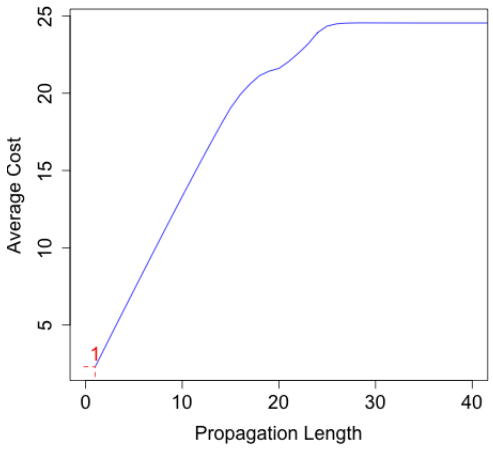


FIGURE 3 – Coût moyen en fonction de la longueur de propagation pour l'instance val8-13.

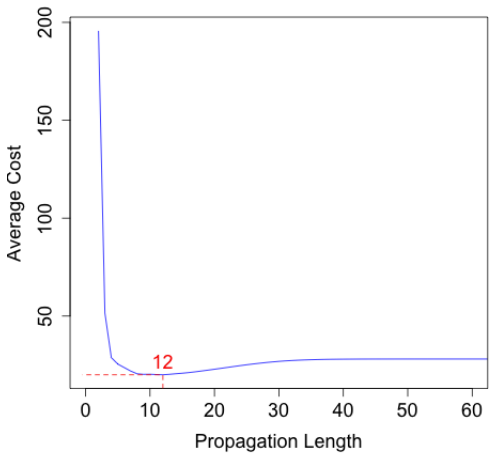


FIGURE 4 – Coût moyen en fonction de la longueur de propagation pour l'instance langford-3-12.

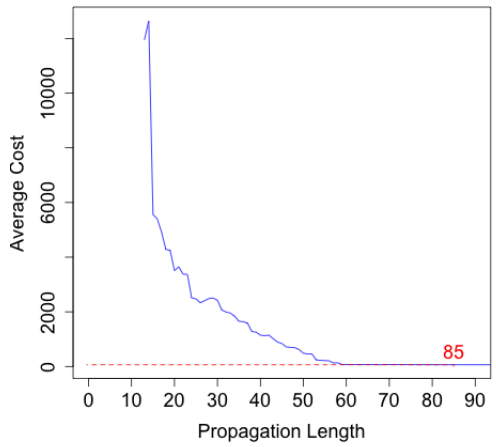


FIGURE 5 – Coût moyen en fonction de la longueur de propagation pour l'instance pseudo-ii32c1.

seuil est recalculé en permanence au cours de la recherche en fonction des résultats précédemment obtenus. Nous calculons le seuil de la façon suivante : nous initialisons la variable $threshold$ à partir des s premiers appels à la fonction GAC^{var} (sans contrôle de la propagation). Autrement dit, au début de la recherche, $threshold$ est initialisé à l'infini. La longueur et le résultat de ces appels sont enregistrés dans une structure de file (FIFO) de taille s. La valeur de seuil ainsi calculée est alors utilisée pour f appels à GAC^{var}. Le $f+1^{ème}$ appel à GAC^{var} est effectué sans contrôle (comme si $threshold$ était positionné à l'infini – ceci n'est pas visible sur l'algorithme). La file va ensuite être mise à jour par simple suppression de la plus ancienne valeur et ajout de la plus récente, ce qui permet de faire évoluer la valeur du seuil. Ce processus est répété cycliquement jusqu'à la fin de la recherche. Pour nos expérimentations, nous avons arbitrairement fixé les valeurs de s à 100, et de f à 10. Les résultats obtenus sont par conséquent préliminaires, et nécessiteront une étude plus poussée de détermination des paramètres en considérant, parmi d'autres pistes, les modèles de séries temporelles (par exemple, les modèles auto-régressifs) utilisés en économétrie [1]. Enfin, remarquons que le fait que $threshold$ soit toujours supérieur ou égal à 1 nous assure de la correction de l'algorithme MAC^c : le niveau de filtrage reste au minimum celui de FC.

5 Résultats expérimentaux

Pour montrer l'intérêt pratique du contrôle de la propagation par contraintes à partir d'une analyse de nature statistique concernant la longueur de propagation, nous avons comparé l'efficacité de MAC, MAC^c et FC pour résoudre de nombreuses séries d'instances CSP. Avec l'utilisation de l'heuristique de choix de variable *dom/wdeg* [6] pour guider la recherche, 1338 instances parmi les 2053 (issues d'un nombre important de séries d'instances) ont été résolues par les trois algorithmes MAC, MAC^c and FC, en moins de 1200 secondes chacune. En moyenne, MAC^c est environ 10% plus rapide que MAC et deux fois plus rapide que FC. Ceci est illustré par le tableau 1 où, pour chaque série, le nombre nb_s d'instances résolues par les trois algorithmes dans le temps imparti (1200 secondes) ainsi que le nombre total nb d'instances sont donnés sous la forme (#Inst=nb_s/nb) en-dessous du nom de la série. Les temps CPU moyens affichés dans ce tableau sont calculés à partir des instances nb_s identifiées pour chaque série ; le nombre d'instances résolues par les trois algorithmes est indiqué à la ligne marquée par #résolus. MAC^c est particulièrement efficace sur les instances de colorations de graphes, et obtient d'assez bons résultats sur les séries pret, et qcp. Le tableau 2 fournit des détails sur la résolution

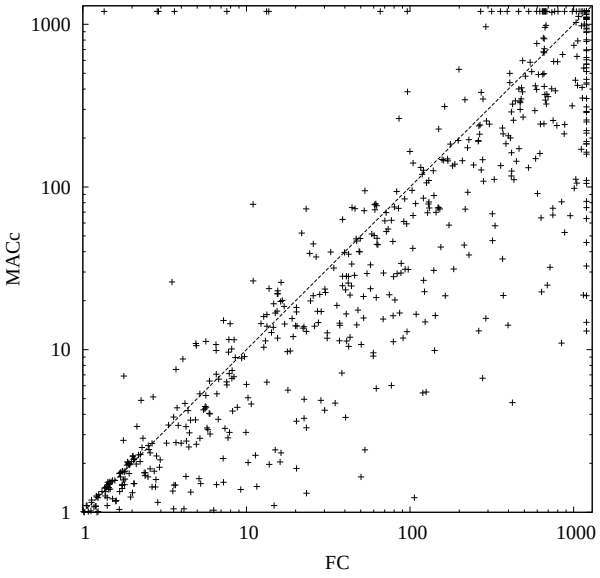
Series		FC	MAC	MAC^c
crosswords-Vg	cpu	92.2	**20.5**	20.7
(#Inst = 200/258)	#résolus	200	214	**217**
graph-coloring	cpu	55.5	58.3	**41.4**
(#Inst = 198/459)	#résolus	210	203	**213**
graph-valiente	cpu	19.9	14.7	**14.2**
(#Inst = 681/793)	#résolus	681	**693**	690
pseudo-ii	cpu	**22.9**	39.1	45.5
(#Inst = 14/41)	#résolus	**31**	16	16
langford	cpu	75.9	**1.06**	1.46
(#Inst = 38/72)	#résolus	38	**47**	**47**
pret	cpu	703	368	**327**
(#Inst = 4/8)	#résolus	4	4	4
qcp	cpu	57.0	48.1	**31.8**
(#Inst = 35/60)	#résolus	**38**	35	**38**
qwh	cpu	45.8	**24.6**	25.3
(#Inst = 30/40)	#résolus	30	30	**31**
rand-3	cpu	190	140	**133**
(#Inst = 120/300)	#résolus	127	**132**	131
scens11	cpu	181	63.6	**59.3**
(#Inst = 9/12)	#résolus	9	**10**	**10**
schurrLemma	cpu	**28.8**	116	50.4
(#Inst = 9/10)	#résolus	9	9	9
All Series	cpu	62.8	36.3	**32.0**
(#Inst = 1338/2053)	#résolus	1387	1393	**1406**

TABLE 1 – Temps moyen (en secondes) pour résoudre les instances des différentes séries (avec un time-out de 1200s pour chaque instance) avec MAC-*wdeg/dom*.

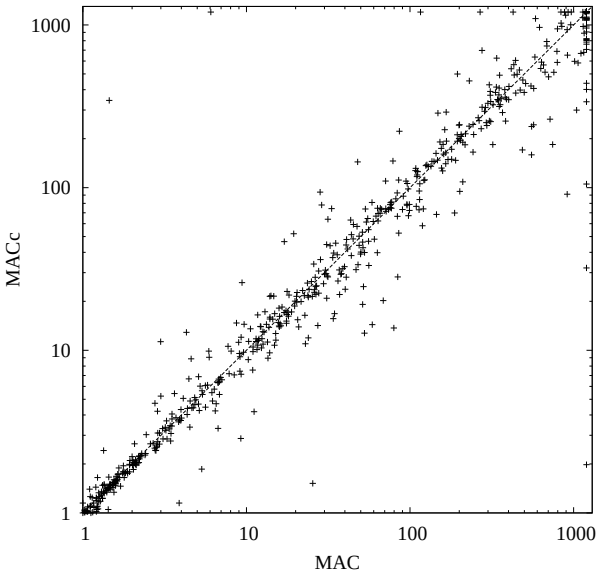
Instances		FC	MAC	MAC^c
cw-words-vg5-9	cpu	308	56.0	**47.0**
	nds	$6,065K$	$127K$	$143K$
2-insertions-5-3	cpu	17.5	**9.7**	**9.7**
	nds	$447K$	$151K$	$163K$
	ccks	$2,149K$	$1,705K$	$1,495K$
david-10	cpu	366	276	**225**
	nds	$9,998K$	$4,439K$	$4,440K$
	ccks	$158M$	$163M$	$148M$
graph-val-8-13	cpu	29.7	27.3	**23.6**
	nds	$826K$	$405K$	$433K$
	ccks	$16M$	$14M$	$11M$
pseudo-ii-32c1	cpu	**7.6**	12.9	14.3
	nds	$120K$	$49,348$	$60,824$
	ccks	$1,107K$	$5,999K$	$6,506K$
langford-3-12	cpu	$1,059$	**13.2**	21.7
	nds	$31M$	$179K$	$345K$
	ccks	$701M$	$4,616K$	$12M$
pret-60-25	cpu	662	354	**323**
	nds	$22M$	$12M$	$10M$
qcp-15-120-5	cpu	62.7	29.9	**21.1**
	nds	$1,742K$	$601K$	$420K$
	ccks	$6,316K$	$7,498K$	$4,617K$
qwh-20-166-0	cpu	157	76.4	**72.6**
	nds	$4,213K$	$1,347K$	$1,348K$
	ccks	$14M$	$17M$	$15M$
rand-3-20-18	cpu	48.6	31.1	**28.6**
	nds	$315K$	$43,952$	$56,085$
scen11-f4	cpu	$1,191$	358	**338**
	nds	$20M$	$3,381K$	$3,718K$
	ccks	$757M$	$261M$	$277M$
lemma-50-9-mod	cpu	**54.0**	202	89.9
	nds	$660K$	$204K$	$204K$
	ccks	$86M$	$706M$	$280M$

TABLE 2 – Résultats détaillés sur diverses instances (time-out de 1200 secondes par instance) avec MAC-*dom/wdeg*.

de diverses instances de la série. Pour chaque instance, le temps CPU total pour la résoudre est donné ainsi que le nombre de noeuds explorés (nds) et le nombre (ccks) de tests de contraintes (sauf pour les contraintes non-binaires en extension où STR2 [13] est utilisé pour appliquer GAC). En général, lorsque MAC et MAC^c explorent à peu près le même arbre de recherche, MAC^c évite de nombreux tests de contraintes, comme on peut le voir pour david-10, qwh-20-166-0, lemma-50-9-mod. Une autre vision des résultats est donné par la figure 6 qui représente des nuages de points permettant une comparaison par paires pour MAC^c opposé à FC et à MAC.



(a) MAC^c est représenté contre FC.



(b) MAC^c est représenté contre MAC.

FIGURE 6 – Comparaison par paires (temps CPU) sur 2053 instances de diverses séries. Le time-out pour résoudre une instance est 20 minutes.

6 Conclusion

Dans cet article, nous avons montré qu'il existe une corrélation intéressante entre la *longueur* et le *résultat* de la propagation de contraintes. Cette observation nous a permis de faire des prévisions raisonnables quant à la capa-

cité de la propagation de contraintes à détecter une incohérence et, par conséquent, nous a conduit à proposer une variante de l'algorithme de recherche MAC lorsque la propagation est stoppée prématurément (i.e. lorsque la probabilité de retour-arrière à partir du noeud courant est faible). Nous croyons que cette corrélation, observée pour la première fois dans la littérature, ouvre de nombreuses autres perspectives pour améliorer l'efficacité de la résolution de contraintes.

Dans un proche avenir, nous projetons de développer des alternatives à notre mode actuel de calcul des valeurs de seuil par exemple, une procédure qui garantisse que x% d'échecs puisse être identifié (où x est l'objectif fixé par l'utilisateur). Nous aimerions également déterminer si certaines caractéristiques des graphes de contraintes telles que la densité, le diamètre, la longueur caractéristique des chemins et/ou le coefficient de regroupement (clustering) [25], peuvent être directement liées à l'efficacité du contrôle de propagation. D'autre part, nous avons l'intention de combiner l'approche probabiliste d'inférence [20] avec notre approche probabiliste de détection d'incohérence, car elles semblent être orthogonales. Enfin, parmi les perspectives que nous avons identifiées, une dernière piste intéressante est l'étude des techniques de shaving s'appuyant sur des tests singletons à coût d'obtention réduit. Cela est prometteur puisque seul le résultat de la propagation est utile pour le shaving. De manière générale, cela pourrait être lié également à la cohérence d'arc paresseuse[22].

Références

[1] *Handbook of Econometrics*. Elsevier, 1983–2007.

[2] C. Bessiere. Constraint propagation. In *Handbook of Constraint Programming*, chapter 3. Elsevier, 2006.

[3] C. Bessiere, P. Meseguer, E.C. Freuder, and J. Larrosa. On Forward Checking for non-binary constraint satisfaction. *Artificial Intelligence*, 141 :205–224, 2002.

[4] C. Bessiere, J.C. Régin, R. Yap, and Y. Zhang. An optimal coarse-grained arc consistency algorithm. *Artificial Intelligence*, 165(2) :165–185, 2005.

[5] 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.

[6] F. Boussemart, F. Hemery, C. Lecoutre, and L. Sais. Boosting systematic search by weighting constraints. In *Proceedings of ECAI'04*, pages 146–150, 2004.

[7] K. Cheng and R. Yap. An MDD-based generalized arc consistency algorithm for positive and negative table constraints and some global constraints. *Constraints*, 15(2) :265–304, 2010.

[8] A. Chmeiss and L. Sais. Constraint satisfaction problems : Backtrack search revisited. In *Proceedings of ICTAI'04*, pages 252–257, 2004.

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

[10] I.P. Gent, C. Jefferson, I. Miguel, and P. Nightingale. Data structures for generalised arc consistency for extensional constraints. In *Proceedings of AAAI'07*, pages 191–197, 2007.

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

[12] C. Lecoutre. *Constraint networks : techniques and algorithms*. ISTE/Wiley, 2009.

[13] C. Lecoutre. STR2 : Optimized simple tabular reduction for table constraint. *Constraints*, to appear, 2011.

[14] C. Lecoutre and F. Hemery. A study of residual supports in arc consistency. In *Proceedings of IJCAI'07*, pages 125–130, 2007.

[15] C. Lecoutre and R. Szymanek. Generalized arc consistency for positive table constraints. In *Proceedings of CP'06*, pages 284–298, 2006.

[16] O. Lhomme and J.C. Régin. A fast arc consistency algorithm for n-ary constraints. In *Proceedings of AAAI'05*, pages 405–410, 2005.

[17] A.K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1) :99–118, 1977.

[18] A.K. Mackworth. On reading sketch maps. In *Proceedings of IJCAI'77*, pages 598–606, 1977.

[19] J.J. McGregor. Relational consistency algorithms and their application in finding subgraph and graph isomorphisms. *Information Sciences*, 19 :229–250, 1979.

[20] D. Mehta and M.R.C. van Dongen. Probabilistic consistency boosts MAC and SAC. In *Proceedings of IJCAI'07*, pages 143–148, 2007.

[21] D. Sabin and E.C. Freuder. Contradicting conventional wisdom in constraint satisfaction. In *Proceedings of CP'94*, pages 10–20, 1994.

[22] T. Schiex, J.C. Régin, C. Gaspin, and G. Verfaillie. Lazy arc consistency. In *Proceedings of AAAI'96*, pages 216–221, 1996.

[23] K. Stergiou. Heuristics for dynamically adapting propagation. In *Proceedings of ECAI'08*, pages 485–489, 2008.

[24] R.J. Wallace and E.C. Freuder. Ordering heuristics for arc consistency algorithms. In *Proceedings of AI/GI/VI'92*, pages 163–169, 1992.

[25] D.J. Watts and S.H. Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 393 :440–442, 1998.

Couplage des approches par contraintes et par analogie : vers une application à la maintenance d'hélicoptères

A. Codet de Boisse[a,b], **É. Vareilles**[a], **T. Coudert**[b], **P. Gaborit**[a], **M. Aldanondo**[a], **L. Geneste**[b]

[a]Université de Toulouse - Mines d'Albi
Route de Teillet, Campus Jarlard
81013 Albi cedex 09

[b]Université de Toulouse - ENI de Tarbes
47, avenue d'Azereix, BP 1629
65016 Tarbes Cedex

[a]{nom}@mines-albi.fr et [b]{nom}@enit.fr

Abstract

Nous détaillons dans cette communication une des manières de prendre en compte simultanément de la connaissance générale et de la connaissance contextuelle pour aider à la décision. Nous nous rapprochons ainsi d'un processus cognitif fréquemment utilisé lors d'une prise de décision humaine. Dans nos travaux, la connaissance générale est modélisée comme un problème de satisfaction de contraintes (*CSP*) et la connaissance contextuelle par une adaptation d'un raisonnement par analogie (*RàPC*). Le couplage de ces deux approches est réalisé à l'aide de contraintes qualifiées de contextuelles qui permettent de calculer certaines contraintes du *CSP* grâce à des valeurs extraites des cas du *RàPC*. Dans un premier temps, nous posons le contexte et la problématique de nos travaux et présentons brièvement les différents couplages identifiés. Dans un deuxième temps, nous détaillons notre usage des *RàPC* et les contraintes contextuelles qui exploitent simultanément les deux types de connaissances. Cette proposition est illustrée sur un exemple tiré du projet FUI DGE Hélimaintenance R&D.

Introduction

La considération simultanée de connaissances générales et de connaissances contextualisées est un processus cognitif utilisé fréquemment lors d'une prise de décision humaine. En effet, la plupart des problèmes auxquels est confronté l'être humain peuvent être résolus en faisant appel à deux types de connaissances : des connaissances générales que la personne possède sur la situation, et des connaissances sur des situations similaires déjà vécues englobant tous les aspects qui ont été mémorisés. Des travaux tant en cognitique qu'en intelligence artificielle travaillent d'une part sur la représentation de ces deux types de connaissances et d'autre part sur leur utilisation conjointe en aide à la décision. Cette utilisation conjointe se décline généralement soit en extraction de connaissances abstraites d'un ensemble de situations vécues, soit en l'utilisation des connaissances générales pour adapter une situation vécue et mémorisée à une nouvelle situation à résoudre [4] et [14].

Dans cette communication, nous représentons les connaissances générales sous forme de problèmes de satisfaction de contraintes ou *CSP* et les connaissances contextualisées sous forme de cas utilisables par un raisonnement à partir de cas ou *RàPC*. Nous présentons ici nos réflexions sur la prise en compte de connaissances contextualisées issues d'un *RàPC* ajoutées de manière dynamique lors du filtrage d'un *CSP*.

Nous rappelons dans la section 1 les deux principes de formalisation de connaissances que sont les approches *CSP* et les approches *RàPC*. Puis dans la section 2, nous exposons comment il est possible d'associer celles-ci afin de prendre en compte les deux types de connaissances dans un processus d'aide à la décision. Dans la section 3, nous présentons notre extension des *RàPC* qui nous permet de conseiller l'utilisateur sur les décisions à prendre. Dans la section 4, nous exposons plus en détails les contraintes contextuelles

qui permettent d'évaluer certaines connaissances générales du *CSP* grâce à des connaissances contextualisées du *RàPC*.

Nous terminons par illustrer nos propositions sur une application industrielle d'aide à la décision en maintenance d'hélicoptères dans le cadre du projet Hélimaintenance R&D.

1 Capitalisation de connaissances et approches retenues

Dans cette communication, nous représentons les connaissances générales sous forme de problèmes de satisfaction de contraintes ou *CSP* et les connaissances contextualisées sous forme de cas utilisables par un raisonnement à partir de cas ou *RàPC*.

1.1 Connaissances générales et *CSP*

Selon [11], les *CSP* sont définis comme un triplet $(\mathbb{V}, \mathbb{D}, \mathbb{C})$ où $\mathbb{V} = \{v_1, v_2, ..., v_k\}$ est un ensemble fini de variables, $\mathbb{D} = \{d_1, d_2, ..., d_k\}$ est un ensemble fini de domaines de définition des variables, et $\mathbb{C} = \{c_1, c_2, ..., c_m\}$ est un ensemble fini de contraintes portant sur les variables où une contrainte décrit les combinaisons autorisées ou exclues des valeurs des variables. De nombreux auteurs tels que [5] [12] [6] ont démontré que ce type de modèles pouvait permettre l'aide à la décision dans le cas d'opérations routinières pour lesquelles des connaissances générales ou expertes peuvent être extraites, explicitées et formalisées.

Réaliser un outil d'aide à la décision à partir des concepts de *CSP* revient à traduire les connaissances soit sous forme d'élément unique de $\mathbb{C}$, soit sous forme de plusieurs éléments répartis sur le triplet $(\mathbb{V}, \mathbb{D}, \mathbb{C})$ [17]. Le modèle de connaissances est alors confondu avec le *CSP*. Trouver une solution d'un problème donné revient à instancier toutes les variables en respectant toutes les contraintes.

Il existe plusieurs manières de traiter un *CSP*. Nous pouvons vouloir :

- réduire les domaines des variables en supprimant les valeurs ne menant pas à une solution,
- trouver une ou toutes les solutions,
- trouver la solution optimale, ou du moins une bonne solution étant donné un objectif portant sur quelques variables ou toutes les variables.

Dans le cadre de nos travaux, nous souhaitons répercuter les choix utilisateur sur le problème courant en éliminant les valeurs devenues incohérentes de manière interactive : c'est la séquence de choix cohérents qui conduit à une ou plusieurs solutions. Nous utilisons donc des techniques de filtrage pour effectuer des déductions sur le problème en détectant les affectations partielles localement ou totalement incohérentes [1]. L'une des techniques les plus utilisées est la technique de renforcement de la cohérence locale ou cohérence d'arc [11] [10]. La cohérence d'arc vérifie que toute valeur du domaine d'une variable est compatible avec chaque contrainte prise séparément.

En pratique, un choix s'exprime comme une réduction du domaine d'une variable. Par application des contraintes liées à cette variable, cette réduction peut réduire le domaine des valeurs possibles des autres variables liées par ces contraintes. Et de proche en proche, tous les domaines peuvent se retrouver réduits. Lorsqu'on ne peut plus déduire aucune réduction de domaine supplémentaire, l'opération de filtrage est terminée. La réduction des domaines possibles pour chacune des variables guide ainsi l'utilisateur pour effectuer son choix suivant. Les itérations successives de ce processus l'amène à converger peu à peu vers une solution.

1.2 Connaissances contextuelles et *RàPC*

Le concept de raisonnement à base de cas ne nécessite pas d'extraction à proprement parler de connaissances. Celles-ci sont regroupées dans les instances du problème appelées « cas ». Chaque cas est caractérisé par un ensemble de valeurs de descripteurs qui va permettre de le différencier des autres cas présents dans la base et de le comparer ensuite à un nouveau problème. Cette comparaison est dans un premier temps faite localement descripteur par descripteur à l'aide d'une fonction de similarité locale. Puis, dans un deuxième temps, une mesure de distance ou similarité globale agrégeant la similarité locale de chaque descripteur, permet de classer les différents cas par rapport à leur similitude avec le problème soumis [9]. Le résultat fourni est un classement par ordre de similarité de l'ensemble des cas. Il faut alors sélectionner un ou plusieurs d'entre eux, les analyser et les adapter pour répondre au problème courant. La base est ensuite enrichie par le problème courant adapté et révisé, décrit au travers des descripteurs. Nous pouvons donc formaliser un *RàPC* comme un triplet $(\mathbb{D}, \mathbb{L}, \mathbb{F})$ où $\mathbb{D} = \{d_1, d_2, ..., d_k\}$ est un ensemble fini de descripteurs, $\mathbb{L} = \{l_1, l_2, ..., l_k\}$ est un ensemble fini de similarité locale, une par descripteur, et $\mathbb{F} = \{f_1, f_2, ..., f_m\}$ est un ensemble fini de fonctions de similarité globale.

Dans les systèmes à base de cas, les connaissances regroupées et noyées dans les cas n'ont pas besoin d'être explicitées pour être exploitées. Seule une description précise des cas est suffisante. Ces connaissances sont

1. Une affectation est partielle lorsque seul un sous-ensemble des variables est instancié.

contextuelles et plus riches que des connaissances générales. En conséquence, elles sont plus idiosyncrasiques et plus difficilement transposables dans un autre contexte ou une autre situation.

Dans le cadre de nos travaux, nous souhaitons conseiller l'utilisateur en nous basant sur des solutions déjà rencontrées et lui proposer un ensemble de valeurs possibles pour chaque descripteur d_i. Par conséquent, nous n'utiliserons que la partie recherche de cas, basée sur les fonctions de similarités locales et globale, des approches *RàPC*. Nous ajoutons à cette recherche de cas similaires, une fonction d'agrégation de valeurs, spécifique à chaque descripteur. Cette fonction permet, à partir de la liste des cas jugés les plus similaires, de proposer à l'utilisateur l'ensemble des valeurs renseignées pour ceux-ci, descripteur par descripteur et ainsi de le conseiller sur ses choix futurs. Nous nommons *EBA* pour *Experience Based Advice* cette utilisation spécifique du *RàPC*.

2 Présentation des types de couplage identifiés

Sept types de couplage possibles entre les approches par contraintes *CSP* et par analogie *RaPC* ont été identifiés [3] et parfois étudiés. Nous en faisons ici la synthèse :

1. Deux types de couplages visent à valider la connaissance stockée soit dans un *CSP*, soit dans un *RàPC*. Par exemple, les travaux de [7] utilisent des cas pour valider ou invalider les contraintes d'un *CSP*. Un *CSP* peut aussi être utilisé pour qualifier la cohérence de cas d'un *RàPC* par rapport à son modèle de contraintes.
2. Deux types de couplage permettent de créer ou modifier de la connaissance soit dans un *CSP*, soit dans un *RàPC*. En effet, l'analyse des cas d'un *RàPC*, peut conduire à ajouter des contraintes dans le *CSP*. De même, les approches par contrainte peuvent être utilisées pour compléter les cas d'un *RàPC* lorsque ceux-ci présentent des descripteurs non-renseignés.
3. Deux types de couplage utilisent les deux approches de manière séquentielle pour aider à la décision. Les travaux de [14] et [16] utilisent de manière séquentielle les *RàPC* pour identifier les cas les plus similaires au problème courant puis un *CSP* pour aider la phase d'adaptation des solutions retenues. De manière similaire, nous proposons d'utiliser un *CSP* pour limiter les valeurs possibles des descripteurs puis un *RàPC* pour détecter les cas les plus similaires au problème courant et en faire l'adaptation. Ce type de solution a déjà été étudiée dans les travaux de [8] et [2] qui proposent des techniques de solicitation de préférence pour le *RàPC* basées sur un pré-filtrage effectué par un *CSP*.
4. Le dernier type de couplage, objet de cette communication, utilise les deux approches de manière à entrelacer les connaissances générales et contextuelles. Nous définissons un type particulier de contraintes, nommées contraintes contextuelles, permettant d'injecter lors du filtrage d'un *CSP*, des valeurs de variables issues des cas passés. Cette approche se base sur notre extension du *RàPC* ou *EBA* pour fournir des conseils sur les valeurs des descripteurs, en considérant certains d'entre eux comme des contraintes pour le *CSP*. Cette considération est réalisée par une expertise lors de la création ou mise à jour des modèles de connaissance.

Dans la suite de nos travaux, comme l'utilisation de connaissances générales et contextuelles doit être transparente pour notre utilisateur, nous regroupons sous le terme **variable** les variables v_i du *CSP* et les descripteurs d_i de l'*EBA*. Nous définissons alors deux ensembles de variables dont l'intersection n'est pas forcément nulle : l'ensemble des variables apparaissant dans les cas ou CBV et l'ensemble des variables apparaissant dans les contraintes ou CSV.

3 Fonctionnement du moteur de recherche et d'agrégation de cas

3.1 Présentation générale de l'*EBA*

L'*EBA* doit permettre de réaliser deux opérations successives : la recherche et l'identification de cas stockés dans une base de cas puis l'agrégation des valeurs des variables sélectionnées dans les cas.

Pour la recherche et l'identification de cas, le principe est assez similaire à celui utilisé dans un système de *RàPC* sauf que le cas cible n'est pas représenté par un ensemble de couples (variable, valeur unique) mais par un ensemble de couples (variable, domaine cible). Le cas cible est confronté à chaque cas de la base de cas (appelés cas sources) afin d'évaluer leur similarité. Pour chaque variable V_{cible} pour laquelle on souhaite effectuer la recherche, une fonction de similarité permet de calculer la similarité locale entre le domaine des valeurs autorisées pour la variable V_{cible} (noté $\mathcal{D}_{V_{cible}}$) et la valeur de cette variable dans le cas source (il s'agit d'un singleton noté V_{source}). Ensuite, pour chaque cas source, l'ensemble des similarités locales est agrégé afin de calculer une similarité globale entre le cas cible et le cas source (cf. section 3.2). Enfin,

chaque cas source dont la similarité globale est supérieure à un seuil fixé (soit par l'utilisateur, soit par un expert) est sélectionné afin de réaliser la seconde opération de l'*EBA* : l'agrégation des valeurs des variables requises (cf. section 3.3).

In fine, l'*EBA* retourne la liste des variables ainsi que leurs domaines de valeurs agrégées.

3.2 Calcul de similarité

Le calcul de similarité s'effectue de manière globale entre un cas cible et un cas source de la base de cas à partir de mesures locales correspondant aux variables. La similarité locale correspondant à une variable V_{cible} est évaluée en considérant un singleton d'une part (la valeur de V_{cible} dans le cas source : V_{source}), et un domaine d'autre part (le domaine $\mathcal{D}_{V_{cible}}$ de la variable V_{cible}). Classiquement, il suffit de vérifier que $V_{source} \in \mathcal{D}_{V_{cible}}$ pour déterminer que la valeur source correspond exactement au domaine cible. Dans le cas contraire, elle ne correspond pas du tout et se voit donc pénalisée dans le calcul de similarité globale.

Cependant, si pour toutes les variables du cas source $V_{source} \notin \mathcal{D}_{V_{cible}}$, il est souhaitable de ne pas l'écarter trop rapidement en vérifiant d'abord la similarité des valeurs V_{source} avec les valeurs ou les bornes du domaine $\mathcal{D}_{V_{cible}}$ correspondant. En cas de valeurs proches, le calcul de similarité globale en tiendra compte. Nous considérons le cas des variables discrètes, celui des variables continues ainsi que le cas où une valeur V_{source} est absente d'un cas source.

1) Si la variable V_{cible} est discrète (symbolique ou numérique), le calcul de similarité locale correspondant consiste à retenir la similarité maximale entre chaque valeur discrète du domaine $\mathcal{D}_{V_{cible}}$ et le singleton V_{source}. Soit le domaine $\mathcal{D}_{V_{cible}}$ défini par l'ensemble des n valeurs autorisées v_i tel que $\mathcal{D}_{V_{cible}} = \{v_1, v_2, ..., v_n\}$. La similarité locale entre la valeur V_{source} et le domaine $\mathcal{D}_{V_{cible}}$ est donnée par l'équation :

$$sim_{loc}(\mathcal{V}_{source}, \mathcal{D}_{V\,cible}) = \max_i(sim(\mathcal{V}_{source}, v_i))$$

L'expression $sim(V_{source}, v_i)$ fait référence à la similarité entre deux valeurs d'une même variable et dont les valeurs sont comprises entre 0 et 1. Il s'agit d'une connaissance a priori fournie par un expert. Celle-ci peut être formalisée soit à l'aide d'une matrice de similarités, soit à l'aide d'une fonction de similarité comme par exemple une fonction seuil, linéaire, sigmoïde ou encore exponentielle (voir [15] pour un panorama).

2) Dans le cas d'une variable continue, trois cas de figures sont possibles. Soit $V_{source} \subset \mathcal{D}_{V_{cible}}$ alors la similarité est de 1 ; soit $V_{source} < \min(\mathcal{D}_{V_{cible}})$ alors la similarité est de $sim(V_{source}, min(\mathcal{D}_{V_{cible}}))$; soit

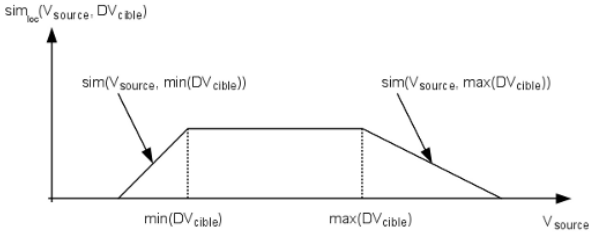


FIGURE 1 – Profil de la similarité entre un domaine $D_{V\,cible}$ et un singleton V_{source} avec variable continue

$V_{source} > max(\mathcal{D}_{V_{cible}})$ alors la similarité est de $sim(V_{source}, max(\mathcal{D}_{V_{cible}}))$. Ceci est représenté par l'équation 1.

Nous illustrons sur la figure 1 un exemple de similarité locale d'une valeur V_{source} par rapport à un domaine $\mathcal{D}_{V_{cible}}$ en utilisant une fonction de similarité linéaire. Cette fonction est utilisées pour évaluer la similarité entre une valeur V_{source} et la borne inférieure ou supérieure du domaine $\mathcal{D}_{V_{cible}}$.

Pour calculer la similarité globale (sim_g), nous utilisons une fonction de Minkowski [15] [1] [13] :

$$sim_g(\mathcal{V}_{source}, \mathcal{D}_{V\,cible}) = 1 - \sum_{i=O}^{Nbvar} [\omega_i * \left(1 - sim_{loc}(\mathcal{V}^i_{source}, \mathcal{DV}^i_{cible})\right)^p]^{1/p}$$

avec :

- $Nbvar$: le nombre de variables à considérer ;
- $\mathcal{V}^i_{source}$: la valeur de la variable i dans le cas source considéré ;
- $\mathcal{DV}^i_{cible}$: le domaine de la variable i dans la cible ;
- ω_i : le poids donné à la variable i.

Cette fonction permet, en faisant varier le paramètre p, de régler le mécanisme d'agrégation. Si $p = 1$, il s'agit d'une moyenne des similarités locales, pondérées par les poids ω_i. Si $p = 2$, l'agrégation fonctionne comme un calcul de distance Euclidienne. Si p devient grand, la fonction tend vers l'opérateur max.

3) Dans le cas où, pour une variable de la cible, la valeur V_{source} est absente d'un cas source, nous distinguons deux situations :

- la valeur n'a pas été renseignée par manque d'informations. Cela signifie donc que ce cas n'est pas complet et dans cette situation, nous considérons que la similarité locale est égale à 1. En effet, lorsque certaines informations sont manquantes, la valeur source peut être potentiellement égale à la valeur cible ;
- la valeur n'a pas été renseignée car elle n'avait pas de raison d'exister dans ce cas (exemple du carburateur qui n'est pas présent dans un moteur diesel). Dans cette situation, nous ne prenons pas

$$sim(\mathcal{V}_{source}, \mathcal{D}_{Vcible}) = \begin{cases} 1 & \text{si} \quad \mathcal{V}_{source} \subset \mathcal{D}_{Vcible} \\ sim(\mathcal{V}_{source}, min(\mathcal{D}_{Vcible})) & \text{si} \quad \mathcal{V}_{source} < min(\mathcal{D}_{Vcible}) \\ sim(\mathcal{V}_{source}, max(\mathcal{D}_{Vcible})) & \text{si} \quad \mathcal{V}_{source} > max(\mathcal{D}_{Vcible}) \end{cases} \quad (1)$$

en compte cette variable dans le calcul de similarité globale.

3.3 Agrégation de valeurs

La seconde opération réalisée par l'*EBA* permet d'agréger les valeurs de chaque variable à partir de l'ensemble des cas identifiés comme suffisamment similaires au cas cible. Deux situations sont donc prises en compte :

- celle de variable symbolique : l'ensemble des valeurs présentes dans les cas retenus est retourné ;
- celle de variable numérique : l'intervalle englobant l'ensemble des valeurs des cas sélectionnés est retourné.

À la fin de cette étape d'agrégation, il est possible de fournir à l'utilisateur des conseils basés sur les expériences passées.

4 Contraintes contextuelles

Les contraintes contextuelles permettent d'utiliser simultanément de la connaissance générale et de la connaissance contextualisée pour aider à la décision. Celles-ci permettent d'injecter lors du filtrage d'un *CSP*, des domaines de valeurs pour certaines variables, domaines issus des cas passés. Cette approche se base sur l'*EBA* pour proposer des conseils sur les domaines de valeurs des descripteurs. Dans le cas des contraintes contextuelles, la recherche des cas similaires est réalisée sur un sous-ensemble des variables de CBV. Certains domaines de valeurs sont ensuite considérés comme des contraintes unaires pour le *CSP*. Cette considération est réalisée à l'aide d'une expertise lors de la création ou mise à jour des modèles de connaissance.

4.1 Définition des contraintes contextuelles

Une contrainte contextuelle contient cinq paramètres :

$LVRe$ **:** La liste des variables de recherche $\subseteq CBV$, c'est-à-dire l'ensemble des variables sur lesquelles la recherche doit être effectuée dans l'*EBA*.

$LVRo$ **:** Les variables de retour de l'*EBA* $\subseteq CBV$, c'est-à-dire les variables dont on souhaite utiliser le domaine de valeurs dans une contrainte du *CSP*.

LVI **:** La liste des variables du *CSP* impactées par la contrainte contextuelle $\subseteq CSV$.

FC **:** La contrainte elle-même, composée des variables de LVI et de $LVRo$. Si plusieurs variables sont à la fois présentes dans le *CSP* et l'*EBA* ainsi que dans les ensembles $LVRe$ et LVI, nous indiçons leur domaines respectifs par le nom de l'ensemble considéré : V_{LVI} ou V_{LVRo}.

ms **:** La similarité minimale, c'est en fait un seuil endessous duquel la similarité entre la cible (les valeurs cherchées) et la source (les valeurs dans la base de cas) est trop faible pour que les cas soient considérés comme similaires.

Une contrainte contextuelle est donc représentée par un quintuplet $cc(LVRe, LVRo, LVI, FC, ms)$

4.2 Filtrage des contraintes contextuelles

Les contraintes contextuelles sont filtrées si l'un des deux évènements suivants apparaît :

- le contexte de recherche de l'*EBA* est modifié. Le contexte de recherche est défini par l'ensemble des variables de CBV réduit. Si l'une des variables de recherche de CBV est réduite par l'utilisateur ou par propagation de contraintes, le contexte de recherche de solution est modifié. Cette réduction va donc préciser le contexte et potentiellement réduire le nombre de cas à prendre en compte pour conseiller l'utilisateur. La réduction du nombre de cas à prendre en compte pourra avoir un impact sur l'évaluation des contraintes contextuelles du *CSP* et donc sur les domaines des variables de CSV.
- l'espace de solutions du *CSP* est modifié. Si l'une des variables de CSV est réduite, cette modification doit être propagée sur l'ensemble des variables liées à elle par une contrainte. Cette réduction de domaines va donc réduire l'espace de solutions défini par le *CSP*. Si cette réduction opère sur l'une des variables de LVI, il faut à nouveau évaluer la contrainte contextuelle par rapport au contexte défini par l'ensemble des variables de $LVRe$ valuées.

L'algorithme 1 représente le fonctionnement global du système étant donné les domaines et les ensembles précédemment définis.

Alg. 1 PROCESSUS D'UTILISATION DU SYSTÈME

– ∴ – *RedV : liste des variables réduites non traitées*
– ∴ – *Vcour : variable courante*
Début
  RedV ← Réduction utilisateur
  Tant Que (*RedV* ≠ ∅) **Faire**
    Vcour ← SHIFT(*RedV*)
    – ∴ – *Si la variable courante appartient à l'ensemble LVRe d'une contrainte contextuelle et a été réduite, le contexte a été modifié. Il faut donc évaluer cette contrainte contextuelle par rapport à ce nouveau contexte.*
    Pour (chaque contrainte contextuelle *cc* dont *LVRe* contient *Vcour*) **Faire**
      Recherche des cas similaires au contexte dans l'*EBA* selon les *LVRe* réduites
      Sélection des cas les plus similaires
      Agrégation des valeurs des variables pour celles appartenant à *LVRo*
      Piltrage par la contrainte *cc*
      – ∴ – *La liste des variables réduites non traitées doit être mise à jour.*
      RedV ← *RedV* ⋃ *LVI* nouvellement réduites
    Fin Pour
    – ∴ – *Si la variable courante appartient au* CSP*, il faut propager cette réduction.*
    Si (*Vcour* ⊂ *CSV*) **Alors**
      Pour (toutes les contraintes *c* portant sur *Vcour*) **Faire**
        Si (*c* est une contrainte contextuelle) **Alors**
          – ∴ – *Ici, le contexte de recherche est défini par l'ensemble des variables de LVRe.*
          Recherche des cas similaires au contexte dans l'*EBA* selon les *LVRe*
          Sélection des cas les plus similaires
          Agrégation des valeurs des variables pour celles appartenant à *LVRo*
        Fin Si
        Filtrage par la contrainte *c*
        – ∴ – *La liste des variables réduites non traitées doit être mise à jour.*
        RedV ← *RedV* ⋃ *CSV* nouvellement réduites
      Fin Pour
    Fin Si
  Fin Tant Que
  – ∴ – *Les domaines des variables sont ici complétement filtrés par les contrainte du* CSP*.*
  – ∴ – *Réévaluation de l'*EBA *pour fournir de nouveaux conseils.*
  Recherche des cas similaires au contexte dans l'*EBA* selon les domaines réduits du *CBV*
  Sélection des cas les plus similaires
  Agrégation des valeurs des variables
  Présentation des domaines conseillés de l'*EBA* et déduits du *CSP* à l'utilisateur
Fin

5 Application à la maintenance d'hélicoptères

Cette section permet d'illustrer sur un cas concret nos propositions : l'estimation du temps et du coût pour de l'aide à la décision en maintenance d'hélicoptère.

5.1 Description du modèle de connaissances

Le modèle se compose de huit variables :

- Durée Théorique DT qui est la durée de maintenance théorique. Cette variable appartient à la fois au *CSP* et à l'*EBA*. Son domaine est défini dans le *CSP*, $D_{DT} = [50, 1000]$ heures et la valeur des cas de l'*EBA* sont cohérents avec cette plage de valeurs.
- Durée Réelle DR qui est la durée réellement passée à effectuer la maintenance. Cette variable n'appartient qu'à l'*EBA* et a pour domaine conseil $Dc_{DR} = [92, 1000]$.
- $Pays$ qui est le pays dans lequel l'hélicoptère a eu l'occasion d'évoluer. Cette variable appartient à la fois au *CSP* et à l'*EBA*. Son domaine est défini dans le *CSP*, $D_{Pays} = \{France, ArabieSaoudite, Norvège, Groenland\}$ et la valeur des cas de l'*EBA* sont cohérents avec cette liste de valeurs.
- $Ambiance$ qui est l'ambiance dans laquelle l'hélicoptère a principalement volé. Cette variable n'appartient qu'à l'*EBA* et a pour domaine conseil $Dc_{Ambiance} = \{Sable, Normal, Froid, Mer\}$.
- $Visite$ qui est le type de visite effectuée (exprimée en heures de vol). Cette variable n'appartient qu'au *CSP* et a pour domaine $D_{Visite} = \{25, 50, 100, 200\}$.
- $PrixMO$ qui est le prix de la main d'œuvre dans le pays dans lequel la maintenance s'effectue (on estime que la maintenance se déroule dans le pays où l'hélicoptère évolue). Cette variable n'appartient qu'au *CSP* et a pour domaine $D_{PrixMO} = \{10, 50, 70, 100\}$.
- Coût Prévisionnel CP qui est le coût théorique de la maintenance de l'hélicoptère. Cette variable n'appartient qu'au *CSP* et a pour domaine $D_{CP} = [0; +\infty[$.
- Durée Prévisionnelle DP qui est la durée prévisionnelle en fonction des aléas dûs à l'ambiance et au pays. Cette variable n'appartient qu'au *CSP* et a pour domaine $D_{DP} = [0; +\infty[$.

Visite	DT
25	[50 ; 150]
50	[150 ; 500]
100	[400 ; 750]
200	[700 ; 1000]

TABLE 1 – Contrainte C1

Pays	Prix MO
France	50
Arabie Saoudite	10
Norvège	100
Groenland	70

TABLE 2 – Contrainte C3

5.2 Modèles *CSP* et *EBA*

5.2.1 Modèle du *CSP*

Le *CSP* est composé de quatre contraintes dont une contrainte contextuelle :

C1 : la contrainte liant le type de visite $Visite$ à sa durée prévisionnelle DT. Cette contrainte est représentée par la table de compatibilité en table 1.

C2 : la contrainte liant le coût prévisionnel de maintenance CP à la durée prévisionnelle DP et au prix de la main d'œuvre $PrixMO$. Cette contrainte est représentée par l'équation 2.

$$CP = DP * PrixMO \tag{2}$$

C3 : la contrainte liant le pays de maintenance $Pays$ au prix de la main d'œuvre $PrixMO$. Cette contrainte est représentée par la table de compatibilité 2.

C4 : la contrainte contextuelle permettant de déduire la durée prévisionnelle DP en fonction de l'ambiance principale de vol $Ambiance$ et du pays $Pays$. Le calcul de la durée prévisionnelle DP se fait en multipliant la durée théorique de maintenance DT par un coefficient déduit du rapport entre les durées réelles déjà constatées DR et les durées théoriques initialement prévues DT enregistrées dans la base de cas. Le seuil de similarité des cas utilisable pour évaluer cette contrainte est fixé à 0.7. Cette contrainte est représentée par le quintuplet :

$$C4(LVRe = \{Ambiance, Pays\},$$
$$LVRo = \{DR, DT\}, LVI = \{DP\},$$
$$FC : DP_{LVI} = DT_{LVI} * \frac{DT_{LVRo}}{DR_{LVRo}}, ms = 0.95)$$

Cas	DT	Pays	Ambiance	DR
1	345	France	Sable	359
2	704	Arabie Saoudite	Normal	844
3	178	Norvège	Froid	182
4	371	France	Sable	484
5	250	France	Normal	250
6	932	Arabie Saoudite	Sable	1000
7	545	Groenland	Sable	575
8	366	France	Normal	366
9	246	Arabie Saoudite	Mer	301
10	671	France	Sable	883
11	631	Arabie Saoudite	Mer	768
12	314	France	Sable	343
13	614	Groenland	Normal	740
14	608	Arabie Saoudite	Sable	867
15	788	France	Sable	904
16	878	Arabie Saoudite	Sable	1000
17	485	Norvège	Normal	485
18	755	Norvège	Mer	875
19	73	Arabie Saoudite	Mer	92
20	726	Arabie Saoudite	Normal	867

TABLE 3 – Base de cas

Ambiance	Normal	Mer	Sable	Froid
Normal	1	0.8	0.7	0.5
Mer		1	0.8	0.2
Sable			1	0.3
Froid				1

TABLE 4 – Similarité pour l'ambiance

5.2.2 Modèle de l'*EBA*

La base de cas utilisée pour l'exemple illustratif est celle représentée dans le tableau 3.

Trois fonctions de similarités locales sont définies dans l'*EBA* :

- Une fonction de similarité locale $sim1$ identique pour les durées réelles DR et théoriques DT qui est représentée par l'équation 3. Cette fonction de similarité locale précise par exemple quelle est la similarité entre une durée source V_{source} de 222 heures et un domaine de recherche ciblé $\mathcal{D}_{V_{cible}}$ de $[260; 356]$: $\frac{222-260+100}{100} = 0.62$.
- La fonction de similarité locale pour l'ambiance $Ambiance$, représentée par la table 4. Cette fonction de similarité locale précise par exemple quelle est la similarité entre une valeur d'ambiance de type Mer et une valeur d'ambiance de type $Froid$: 0.2.
- La fonction de similarité locale pour le pays $Pays$, représentée dans la table 5. Cette fonction de similarité locale précise par exemple quelle est la similarité entre le $France$ et la $Norvège$: 0.5.

$$sim1 = \begin{cases} 1 & \text{si} \quad V_{source} \subset \mathcal{D}_{cible} \\ (V_{source} - min(\mathcal{D}_{cible}) + 100)/100 & \text{si} \quad V_{source} < min(\mathcal{D}_{cible}) \\ (-V_{source} - max(\mathcal{D}_{cible}) + 100)/100 & \text{si} \quad V_{source} > max(\mathcal{D}_{cible}) \end{cases} \quad (3)$$

Pays	France	Arabie Saoudite	Norvège	Groenland
France	1	0.6	0.5	0.2
Arabie Saoudite		1	0.2	0
Norvège			1	0.7
Groenland				1

TABLE 5 – Similarité pour l'ambiance

Notre exemple ne comprend qu'une seule fonction de similarité globale :

$$sim_g(\mathcal{V}_{source}, \mathcal{D}_{Vcible}) = 1 - \sum_{i=O}^{4} [1/4 * \left(1 - sim_i(\mathcal{V}^i_{source}, \mathcal{DV}^i_{cible})\right)$$

5.3 Exemples de fonctionnement

Nous illustrons nos propositions sur le modèle précédent. L'état initial du système d'aide à la décision en maintenance est le suivant :

- durée Théorique DT, $D_{DT} = [50, 1000]$,
- durée Réelle DR, $Dc_{DR} = [92, 1000]$,
- $Pays$, $D_{Pays} = \{France, ArabieSaoudite, Norvège, Groenland\}$,
- $Ambiance$, $Dc_{Amb} = \{Sable, Normal, Froid, Mer\}$,
- type de visite $Visite$, $D_{Visite} = \{25, 50, 100, 200\}$,
- prix de la main d'œuvre $PrixMO$, $D_{PrixMO} = \{10, 50, 70, 100\}$,
- Coût Prévisionnel CP, $D_{CP} = [0; +\infty[$,
- Durée Prévisionnelle DP, $D_{DP} = [0; +\infty[$.

5.3.1 Sélection de Visite

Dans un premier temps, l'utilisateur fixe la valeur de *Visite* à 200. Cette variable appartenant à CSV, il faut propager cette réduction. La contrainte **C1** est donc propagée et la valeur de DT est réduite à $[700; 1000]$.

La contrainte contextuelle **C4** est ensuite propagée car la variable DT appartient à cette contrainte. Une recherche dans l'*EBA* est donc effectuée sur le nouveau contexte. Étant donné que le contexte de recherche de la contrainte **C4** porte sur les variables *Ambiance* et *Pays* qui ne sont réduites, l'ensemble des cas de la base est remonté avec une similarité supérieure à 0.7.

Les ensemble englobant les valeurs de DR_{LVRo} et de DT_{LVRo} sont donc agrégés et remontés : $D_{DR_{LVRo}} = [92, 1000]$ et $D_{DT_{LVRo}} = [73, 932]$. La contrainte **C4** peut donc être propagée, entraînant ainsi $DP = (DR_{LVI} \otimes (DT_{LVRo} \oslash) DR_{LVRo}) \cap DP = ([700; 1000] \otimes ([73; 932] \oslash [92; 1000])) \cap [0; +\infty[= [51.1; 10130.4]$.

DP ayant été modifiée, la contrainte **C2** est alors propagée, entraînant $CP = DP \otimes MO = ([51.1; 10130.4] \otimes \{10; 50; 70; 100\}) \cap [0; +\infty[= [511; 1013043.47]$.

5.3.2 Sélection de Pays

$Visite$ reste fixé à 200, entraînant l'ensemble des déductions faites en section 5.3.1. L'utilisateur vient alors à fixer $Pays$ à *Norvège*.

La contrainte **C3** est tout d'abord propagée entraînant $PrixMO = 100$. La contrainte **C4** est propagée car $Pays \in LVRe$ a été valuée. Les cas $3, 17$ et 18 sont donc sélectionnés car le contexte de la recherche ne porte que sur la variable $Pays = Norvège$. Les ensembles englobant les valeurs de DR_{LVRo} et de DT_{LVRo} sont donc agrégés et remontés : $D_{DR_{LVRo}} = [182; 875]$ et $D_{DT_{LVRo}} = [178; 755]$. La contrainte **C4** peut donc être propagée, entraînant ainsi $DP = (DR_{LVI} \otimes (DT_{LVRo} \oslash) DR_{LVRo}) \cap DP = ([700; 1000] \otimes ([178; 755] \oslash [182; 875])) \cap [51.1; 10130.4] = [142.4; 4148.35]$.

DP ayant été modifiée, la contrainte **C2** est propagée, entraînant $CP = DP \otimes MO = ([142.4; 4148.35] \otimes \{100\}) \cap [511; 1013043.47] = [14240; 414835.16]$.

La variable $Pays$ appartenant à CBV, le contexte a été modifié. Une mise à jour des domaines conseils doit donc être réalisée. La recherche de cas similaires avec un seuil de 0.7 sélectionne les cas $1, 3, 4, 7, 8, 10, 12, 13, 15, 17$ et 18 de la base de cas, présentée par la table 3 et pour lesquels la mesure de similarité est supérieure au seuil fixé. La fonction d'agrégation de valeurs fournit donc les domaines conseils suivants : $DT = [178; 788], Pays = \{Norvège\}, Ambiance = \{Sable, Froid, Mer, Normal\}$ et $DR = [182; 904]$.

5.3.3 Sélection de Ambiance

$Visite$ et $Pays$ restent fixés et les valeurs restent les mêmes qu'à la fin de la section 5.3.2. L'utilisateur fixe alors $Ambiance$ à $Froid$.

La variable $Ambiance$ appartenant à $LVRe$ de la contrainte **C4**, celle-ci doit être propagée. Seul le cas 3 est sélectionné car le contexte de la recherche porte sur les variables $Pays = Norvège$ et $Ambiance = Froid$. L'ensemble englobant les valeurs de DR_{LVRo} et de DT_{LVRo} sont donc agrégés et remontés : $D_{DR_{LVRo}} = 182$ et $D_{DT_{LVRo}} = 178$. La contrainte **C4** peut donc être propagée, entraînant ainsi $DP = (DR_{LVI} \otimes (DT_{LVRo} \oslash) DR_{LVRo}) \cap DP = ([700; 1000] \otimes (178 \oslash 182)) \cap [142.4; 4148.35] = [84.61; 978.021]$.

DP ayant été modifiée, la contrainte **C2** est propagée, entraînant $CP = DP \otimes MO = ([84.61; 978.021] \otimes \{100\}) \cap [14240; 414835.16] = [8461.5384615385, 97802.197]$.

La variable $Ambiance$ appartenant à CBV, le contexte a donc changé. Une mise à jour des domaines conseils doit être réalisée. La recherche de cas similaires avec un seuil de 0.7 sélectionne les cas $3, 7$ et 17 de la base de cas présentée par la table 3. La fonction d'agrégation de valeurs fournit donc les domaines suivants : $DT = [178; 545], Pays = \{Norvège\}, Ambiance = \{Froid\}$ et $DR = [182; 575]$.

5.3.4 Estimation de la durée théorique DT

$Visite$, $Pays$ et $Ambiance$ restent fixés et les valeurs restent les mêmes qu'à la fin de la section 5.3.3. L'utilisateur estime alors la durée théorique de maintenance DT à 700. Nous aboutissons à l'état final suivant :

- durée Théorique DT, $D_{DT} = 700$,
- durée Réelle DR, $Dc_{DR} = 182$,
- $Pays$ $D_{Pays} = \{Norvège\}$,
- $Ambiance$, $Dc_{Amb} = \{Froid\}$,
- type de visite $Visite$, $D_{Visite} = \{200\}$,
- prix de la main d'œuvre $PrixMO$, $D_{PrixMO} = \{100\}$,
- Coût Prévisionnel CP, $D_{CP} = 68461.53$,
- Durée Prévisionnelle DP, $D_{DP} = 684.615$.

Conclusions et perspectives

Le but de cette communication était de présenter une manière de prendre en compte simultanément des connaissances générales et des connaissances contextuelles dans un unique système d'aide à la décision. Les connaissances générales sont modélisées comme un *CSP* tandis que les connaissances contextuelles dans une base de cas.

Dans un premier temps, nous avons rappelé les deux principes de capitalisation et de formalisation de la connaissance utilisés. Puis, nous avons fait la synthèse des différentes manières de coupler ces deux approches pour utiliser les deux types de connaissances dans un processus d'aide à la décision.

Dans un troisième temps, nous avons proposé une utilisation spécifique du *RàPC*, nommée *EBA*, qui nous permet de fournir des conseils à l'utilisateur basés sur les cas passés. Pour ce faire, nous utilisons le principe de recherche de cas similaires du *RàPC* étendue à des domaines cibles, basée sur des fonctions de similarités locales et globales. Nous ajoutons à ce principe de recherche la notion de seuil de similarité qui nous permet de ne retenir que les cas les plus similaires à notre recherche. Une fonction d'agrégation de valeurs est aussi proposée afin de fournir à l'utilisateur l'ensemble des valeurs renseignées pour les cas jugés les plus similaires comme domaines de conseils.

Dans un quatrième temps, nous définissons la notion de contrainte contextuelle. Ce type particulier de contrainte permet d'injecter les domaines de conseils de certaines variables comme domaines dans contraintes du *CSP*. Cette notion repose sur une hypothèse faite par un expert de cohérence entre les connaissances stockées dans le *CSP* et l'*EBA*. La définition des contraintes contextuelles est posée de manière formelle et le processus complet d'utilisation de notre système d'aide à la décision basée sur des connaissances générales et contextuelles est défini.

Dans un dernier temps, un exemple tiré de notre problématique industrielle illustre nos propositions.

Nous envisageons de recueillir des préférences émises par l'utilsateur sous forme de domaines pour améliorer les conseils fournis par l'*EBA*. Par ailleurs, pour aider l'expert à découvrir dans la base de cas des groupes de variables qui pourraient donner lieu à de nouvelles contraintes contextuelles et pour valider l'expression de cette contrainte, nous envisageons d'utiliser des techniques d'analyse de données et statistiques.

Remerciements

Les auteurs tiennent à remercier tous les partenaires du projet Hélimaintenance R&D, en particulier les experts d'*IXAIRCO* pour leur engagement dans la construction des modèles de *CSP* et *EBA*.

Références

[1] Y. Avramenko and A. Kraslawski. Similarity concept for case-based design in process engineering. *Elsevier - Computers and Chemical Engineering*, 30 :548–557, 2006.

[2] C. Boutilier, R. Brafman, C. Geib, and D. Poole. A constraint-based approach to preference elicitation and decision making. In *AAAI Spring Symposium on Qualitative Decision Theory*, page 14, 1997.

[3] Aurélien Codet de Boisse, Elise Vareilles, Michel Aldanondo, Paul Gaborit, Thierry Coudert, and Laurent Geneste. Couplage csp et cbr : premières identifications des modes de couplage. In *MOSIM 2010*, Mai 2010.

[4] A. Didierjean. Apprendre à partir d'exemples : extraction de règles et/ou mémoires d'exemplaires ? *Année psychologique*, 101 :325–348, 2001.

[5] S. Dutta, Wieranga B., and Dalebout A. Designing management support systems using an integrative perspective. *Communications of the ACM*, 40, Issue 6 :70–79, 1997. ISSN 0001-0782.

[6] H. Fargier, J. Lang, and T Schiex. Mixed constraint satisfaction : a framework for decision problems under incomplete knowledge. In *AAAI-96*, pages 175–180, 1996.

[7] Alexander Felfernig, Gerhard Friedrich, Klaus Isak, Kostyantyn Shchekotykhin, Erich Teppan, and Dietmar Jannach. Automated debugging of recommender user interface descriptions. *Applied Intelligence*, 31(1) :1–14, 2007.

[8] V. Ha and P. Haddawy. Toward case-based preference elicitation : Similarity measures on preference structures. In *Proceedings of the Fourteenth Conference on Uncertainty in Artificial Intelligence*, page 193–201, 1998.

[9] J. Kolodner. *Case-based reasoning*. Morgan Kaufman Publishers, 1993.

[10] O. Lhomme. Consistency techniques for numeric csp. *International Joint Conference on Artificial Intelligence*, 13 :232–238, 1993.

[11] U. Montanari. Networks of constraints : fundamental properties and application to picture processing. In *Information sciences*, volume 7, pages 95–132, 1974.

[12] H.R. Nemati, Steiger D. M., L. S. Iyer, and R. T et Herschel. An architectural integration of knowledge management. *Decision Support Systems*, 33 :143–161, 2002. ISSN 0167-9236, DOI : 10.1016/S0167-9236(01)00141-5.

[13] H. Nunez, M. Sanchez-Marre, U. Cortes, J. Comas, M. Martnez, I Rodrıguez-Roda, and M. Poch. A comparative study on the use of similarity measures in casebased reasoning to improve the classification of environmental system situations. *Elsevier - Environmental Modelling and Software*, 19 :809–819, 2004.

[14] Lisa Purvis and Pearl Pu. National science foundation grant iri. Technical report, Adaptation Using Constraint Satisfaction Techniques, 1995.

[15] Bergmann R. Experience management : Foundations, development methodology and, internet-based applications. *Springer-Verlag Berlin Heidelberg*, 2002.

[16] Magali Ruet and Laurent Geneste. Search and adaption in a fuzzy object oriented case base. In *Advances in Case-Based Reasoning*. ECCBR, September 2002.

[17] Y. Vernat. *Formalisation et qualification de modèles par contraintes en conception préliminaire*. Thèse de doctorat, École Nationale des Arts et Métier, Bordeaux, France, 2004.

Enumérer tous les modèles de formules booléennes par poids croissant*

Nadia Creignou **Frédéric Olive** **Johannes Schmidt**

Laboratoire d'Informatique Fondamentale de Marseille,
CNRS UMR 6166 - Aix-Marseille Université
{nadia.creignou,frederic.olive,johannes.schmidt}@lif.univ-mrs.fr

Résumé

Nous nous intéressons à l'énumération des modèles de formules booléennes par poids croissant. Le poids d'un modèle est le nombre de variables assignées à 1. Un algorithme est considéré comme efficace dans ce contexte s'il énumère les modèles l'un après l'autre dans le bon ordre en maintenant un délai polynomial entre deux solutions successives. Nous nous plaçons dans le cadre de Schaefer. L'énumération des modèles a déjà été abordée dans ce cadre, mais sans prise en compte de l'ordre d'énumération. Cette contrainte a une incidence cruciale sur la complexité. Nous obtenons une nouvelle classification dichotomique de la complexité de ce problème. D'une part nous présentons de nouveaux algorithmes à délai polynomial pour énumérer les modèles des formules de Horn et des formules 2-XOR par poids croissant ; d'autre part, nous prouvons que pour les formules ne relevant pas de l'un des deux types susmentionnés, il n'existe pas de tels algorithmes à délai polynomial, à moins que P = NP.

1 Introduction

Cet article traite de l'algorithmique et de la complexité de l'*énumération*, c'est-à-dire de la génération de toutes les solutions d'un problème donné. Cette question suscite un intérêt croissant depuis une dizaine d'années, du fait de l'explosion de la taille des données traitées par les ordinateurs dans les applications les plus quotidiennes. Pensons par exemple aux requêtes dans des bases de données, qui admettent fréquemment des volumes énormes de réponses. Présenter ces réponses de manière incrémentale efficace est un enjeu d'importance croissante. Ainsi, un utilisateur de moteur de recherche attend que les premiers résultats d'une recherche par mots clé soient instantanément produits. Au titre des autres champs d'application de l'énumération, citons notamment la résolution de contraintes, la recherche opérationnelle, la fouille de données, l'exploration du web, la bioinformatique et la linguistique computationnelle (voir e.g. [17, 8, 1]).

Du fait de la quantité de solutions possiblement générées par les algorithmes d'énumération, la taille de leur sortie est souvent bien plus importante que celle de leur entrée. Par conséquent, le temps polynomial mesure mal la performance de tels algorithmes, et on préfère quantifier la *régularité* de ces types de processus, plutôt que la durée totale de leur exécution. Ainsi, c'est la notion de *délai polynomial* qui est habituellement acceptée comme formalisation de la "faisabilité", en matière d'algorithmes d'énumération : un algorithme d'énumération est dit *à délai polynomial* si le temps écoulé entre deux solutions successives est polynomial en la taille de l'entrée.

Depuis le résultat inaugural de Schaefer [19], le point de vue des problèmes de satisfaction de contraintes (CSP) sur la théorie de la complexité a fait la preuve de sa pertinence (voir par exemple [5]). En offrant un cadre unifié intimement lié à divers problèmes issus des bases de données, il a permis de nombreux résultats de classification relatifs à différentes notions de complexité (voir [7] pour une synthèse). Le présent papier s'inscrit dans cette ligne de recherche.

Pour définir un CSP booléen non uniforme, on fixe un ensemble fini de relations booléennes Γ, appelé langage de contraintes. Une Γ-formule est alors une conjonction de clauses dont la forme est dictée par Γ. Le problème de l'énumération se formule alors comme suit : une Γ-formule étant fixée, peut-on énumérer efficacement tous ses modèles ? Dans [4], Creignou et

*Accepté à SAT'11. Avec le soutien de l'ANR *ENUM* 07-BLAN-0327-04.

Hébrard ont abordé cette question en prouvant un premier résultat de classification sur l'énumération relative aux Γ-formules : si Γ est Horn, dual-Horn, bijonctive ou affine, il existe un algorithme à délai polynomial qui énumère les modèles de toute Γ-formule ; sinon, un tel algorithme n'existe pas, à moins que P = NP.

Mais ce dernier résultat ignore un aspect important de la conception des algorithmes d'énumération : la spécification de l'ordre dans lequel les solutions sont générées. Or cet aspect est déterminant : dans de nombreux cas, on ne peut pas se permettre de générer l'intégralité des solutions, et l'on cherche plutôt à produire les plus "importantes" relativement à une *métrique* particulière, incarnée par cet ordre. Dans d'autres cas, on cherche une solution satisfaisant des propriétés supplémentaires complexes, et on génère les solutions par *ordre de préférence* jusqu'à en trouver une acceptable (voir quelques exemples dans [15, 26]). Par ailleurs, il s'avère que l'ordre imposé dans un problème d'énumération affecte profondément la complexité du problème. Johnson et al. [10] ont ainsi prouvé que les indépendants maximaux d'un graphe peuvent être énumérés dans l'ordre lexicographique par un algorithme à délai polynomial, alors qu'un tel algorithme n'existe pas pour l'ordre anti-lexicographique, à moins que P = NP.

Dans ce papier, nous spécifions l'ordre d'énumération. Nous traitons de l'énumération des modèles d'une formule *relativement à leur poids*, c'est-à-dire aux nombre de variables qu'ils évaluent à 1. Le poids est un paramètre naturel dans les CSP booléens [16, 18, 6], qui peut être assimilé au *coût* du modèle. Notre point de vue renvoie de ce fait aux nombreux travaux sur l'énumération par coût décroissant [25, 15, 26]. Ainsi, la question centrale de ce papier est : peut-on énumérer efficacement les modèles d'une Γ-formule par poids croissant ? Nous répondons à cette question par un résultat de classification dichotomique : si l'ensemble Γ de relations est Horn ou Krom, il existe un algorithme à délai polynomial qui énumère les modèles de toute Γ-formule par poids croissant ; sinon, un tel algorithme n'existe pas, à moins que P = NP. La preuve de ce résultat s'appuie sur de nouveaux algorithmes d'énumération pour les CSP booléens, en particulier pour les formules de Horn.

L'article est organisé comme suit : dans la Section 2, nous rappelons le vocabulaire de base des CSP booléens et de l'énumération. Nous énonçons aussi notre résultat principal, le Théorème 2.2. La preuve de ce théorème est détaillée au cours des sections suivantes. La Section 3 présente des algorithmes d'énumération efficaces, dans les cas où Γ est Krom ou Horn. Dans la Section 4, nour prouvons le versant négatif du Théorème 2.2 : pour toute relation Γ qui n'est ni Horn ni Krom, l'existence d'un algorithme à délai polynomial pour l'énumération des modèles d'une Γ-formule par poids croissant impliquerait P = NP. Nous concluons et pointons quelques questions ouvertes dans la Section 5.

2 Matériel

2.1 Langages de contraintes et Γ-formules

Une *relation logique* d'arité k est une relation $R \subseteq \{0,1\}^k$. Par abus de notation, nous ne différencions pas une relation et son symbole de prédicat. Une *contrainte* C est une formule $C = R(x_1, \ldots, x_k)$, où R est une relation logique k-aire et où les x_i sont des variables non nécessairement distinctes. Si u et v sont deux variables, on note $C[u/v]$ la contrainte obtenue à partir de C en remplaçant chaque occurrence de v par u. Si V est un ensemble de variables, $C[u/V]$ désigne la contrainte obtenue à partir de C par substitution de u à chaque occurrence de toute variable de V.

Une assignation m de valeurs de vérité aux variables *satisfait* la contrainte C si $\big(m(x_1), \ldots, m(x_k)\big) \in R$. Un *langage de contrainte* Γ est une ensemble fini de relations logiques. Une Γ-*formule* φ est une conjonction de contraintes écrites avec les seules relations logiques de Γ. C'est donc une formule du premier ordre sans quantificateur. On note $\mathrm{Var}(\varphi)$ l'ensemble de variables apparaissant dans φ. Une Γ-formule φ est satisfaite par une assignation $m : \mathrm{Var}(\varphi) \to \{0,1\}$ si m satisfait simultanément toutes les contraintes de φ. L'assignation m est alors un *modèle* de φ.

Un ordre canonique sur les variables étant fixé, une assignation s'interprète naturellement comme un ensemble de tuples. L'ensemble des modèles d'une formule φ constitue alors une relation logique R_φ que l'on confondra le plus souvent avec φ elle-même.

Au cours de l'article, nous nous référerons à la terminologie de Schaefer [19] pour classifier les relations booléennes. Une relation booléenne R est *Horn* (resp. *dual-Horn*) si R peut être définie par une formule de Horn (resp., dual-Horn), c'est-à-dire une formule sous forme normale conjonctive (CNF pour abréger), dont chaque clause contient au plus un litéral positif (resp., au plus un litéral négatif). Une relation R est *bijonctive* si elle peut être définie par une formule 2-CNF. Une relation R est *affine* si elle peut être définie par une formule *affine*, i.e. par une conjonction de XOR-clauses (obtenue par un XOR de plusieurs variables plus éventuellement la constante 1). Une telle formule peut aussi être vue comme un système d'équations linéaires sur GF[2]. Une relation est *Krom* (on dit aussi *affine de largeur* 2) si elle est définissable par

une conjonction de clauses, chacune d'entre-elles étant soit une clause unaire, soit une 2-XOR-clause (obtenue par un XOR de 2 variables plus éventuellement la constante 1). Une telle formule peut aussi être vue comme un système d'équations linéaires sur GF[2] avec au plus deux variables par équation. Une relation R est 0-*valide* (resp. 1-*valide*) si $R(0,\ldots,0)=1$ (resp. $R(1,\ldots,1)=1$). Finalement, un langage de contrainte Γ est Horn (resp. dual-Horn, bijonctif, affine, Krom, 0-valide, 1-valide) si chaque relation de Γ est Horn (resp. dual-Horn, bijonctive, affine, Krom, 0-valide, 1-valide). Nous disons qu'un langage de contrainte est *Schaefer* s'il est Horn, dual-Horn, bijonctif, ou affine.

On dispose de critères efficaces pour déterminer l'appartenance d'une relation aux classes pré-citées. Chacune de ces classes est en effet caractérisée par ses polymorphismes (voir e.g. [7] pour une description détaillée). Rappelons brièvement ces propriétés. Les opérations de conjonction, disjonction et addition appliquées sur des k-uplets booléens le sont coordonnée par coordonnée. Ainsi pour $m, m', m'' \in \{0,1\}^k$:

$$\begin{aligned} m \wedge m' &= (m[1] \wedge m'[1], \ldots, m[k] \wedge m'[k]) \\ m \vee m' &= (m[1] \vee m'[1], \ldots, m[k] \vee m'[k]) \\ m \oplus m' &= (m[1] \oplus m'[1], \ldots, m[k] \oplus m'[k]) \end{aligned}$$

où $m[i]$ est la i-ème coordonnée du vecteur m et $\oplus$ est l'opérateur XOR (le "ou exclusif"). Pour une relation logique R, les *propriétés de clôture* suivantes caractérisent la structure de R.

- (R est Horn) ssi $(m, m' \in R \Rightarrow m \wedge m' \in R)$.
- (R est dual-Horn) ssi $(m, m' \in R \Rightarrow m \vee m' \in R)$.
- (R est affine) ssi $(m, m', m'' \in R \Rightarrow m \oplus m' \oplus m'' \in R)$.
- (R est affine et 0-valide) ssi $(m, m' \in R \Rightarrow m \oplus m' \in R)$.

Le problème de la satisfaisabilité pour les Γ-formules, noté $\textsc{Sat}(\Gamma)$, fût d'abord étudié par Schaefer [19] qui établit sa fameuse classification dichotomique : si Γ est Schaefer ou 0-valide ou 1-valide, alors $\textsc{Sat}(\Gamma)$ est dans P ; sinon $\textsc{Sat}(\Gamma)$ est NP-complet. La complexité de la recherche d'une solution non triviale (i.e. d'un modèle qui évalue au moins une variable à 0 et une variable à 1), $\textsc{Sat}^*(\Gamma)$, est traité dans [4] : si Γ est Schaefer, alors $\textsc{Sat}^*(\Gamma)$ est dans P ; sinon, $\textsc{Sat}^*(\Gamma)$ est NP-complet. Depuis, des classifications de la complexité de nombreux autres problèmes algorithmiques relatifs aux Γ-formules ont été obtenues (voir [7] pour une synthèse).

2.2 Enumération

Nous traitons donc ici de l'énumération par poids croissant des modèles de formules booléennes généralisées, le poids d'une assignation étant le nombre de variables qu'il envoie sur 1. Ce problème peut être formulé comme suit :

Problème :	$\textsc{Enum-Sat}_w(\Gamma)$
Entrée :	une Γ-formule φ.
Sortie :	générer tous les modèles de φ par poids croissant.

On dit qu'un algorithme $\mathcal{A}$ calcule le problème d'énumération $\textsc{Enum-Sat}_w(\Gamma)$ si, pour une formule φ donnée, $\mathcal{A}$ génère les modèles de φ l'un après l'autre, par poids croissant, sans doublons, et s'arrête après la dernière sortie.

Le temps polynomial ne fournit pas une évaluation pertinente de l'efficacité des algorithmes d'énumération, parce que la sortie de tels algorithmes est de taille potentiellement exponentielle en la taille de leur entrée. Dans [10] plusieurs notions d'efficacité sont présentées. Le moins que l'on puisse exiger, est que l'énumération s'effectue en *temps total polynomial*, c'est-à-dire que le temps requis pour générer toutes les solutions soit polynomial en la taille de l'entrée *et* en le nombre de solutions (i.e. en la taille de la sortie). Une propriété attendue d'un algorithme d'énumération est qu'il commence à retourner des solutions le plus tôt possible et, plus généralement, qu'il génère les solutions de façon régulière, avec un délai borné entre deux sorties consécutives. On dit qu'un algorithme d'énumération est *à délai polynomial* si le temps écoulé avant la première sortie et le délai entre deux sorties consécutives sont bornés par un polynôme $p(n)$ en la taille de l'entrée. Notons qu'un tel algorithme génère les k premières solutions en temps $k \cdot p(n)$. Ceci est une proprieté importante des algorithmes à délai polynomial, parce que même si l'on ne dispose pas du temps nécessaire à l'énumération de *toutes* les solutions, du moins les k premières solutions seront-elles énumérées rapidement. Si $\textsc{Enum-Sat}_w(\Gamma)$ est calculable par un algorithme à délai polynomial, nous écrivons $\textsc{Enum-Sat}_w(\Gamma) \in \text{delayP}$.

Pour caractériser la complexité en espace, nous ignorons l'espace utilisé pour écrire les sorties, seul l'espace requis pour stocker des résultats intermédiaires est pris en compte. On exige parfois d'un algorithme d'énumération qu'il s'exécute en *espace polynomial*, i.e. que l'espace qu'il utilise pendant tout son calcul soit polynomial en la taille de l'entrée. Cette demande est restrictive, même pour un algorithme à délai polynomial. En effet certains algorithmes à délai polynomial, en particulier quand un ordre spécifique est imposé, s'appuient sur des structures de données de taille exponentielle en celle de la donnée (voir [10, 14, 24]). Ceci vaut notamment pour l'algorithme 2 décrit dans la Section 3.

Pourtant tout algorithme à délai polynomial utilise un *espace polynomial incrémental*. Ceci signifie que

l'espace requis pour générer les k premières solutions est borné par le produit de k et d'un polynôme en la taille de l'entrée.

2.3 Résultat principal

La complexité de l'énumération des modèles de formules booléennes en l'absence d'un ordre préalablement spécifié a été étudiée dans [4]. Une approche alternative, utilisant les polymorphismes partiels, a été développée plus récemment dans [21].

Théorème 2.1 *(*Enumération des modèles [4]*.) Si* Γ *est Schaefer, alors il existe un algorithme à délai polynomial qui génère tous les modèles d'une* Γ*-formule. Sinon, un tel algorithme n'existe pas, à moins que* P = NP.

Dans cet article nous nous intéressons à l'énumération des modèles par poids croissant. Bien sûr, quand Γ est Schaefer, le Théorème 2.1 permet d'exécuter cette tâche en *temps total polynomial*. Il suffit de générer d'abord toutes les solutions via l'algorithme à délai polynomial qui sous-tend la preuve de ce théorème, puis de trier les solutions afin de les produire par poids croissant. Cependant, une telle procédure interdit tout contrôle sur la régularité de l'énumération : s'il y a un nombre exponentiel de modèles, la première solution est produite après un temps exponentiel. Il s'avère nécessaire de développer des techniques plus élaborées pour réaliser une énumération efficace dans cet ordre. Les ensembles de relations qui admettent un bon comportement quant à cette tâche ne coïncident pas avec celles de Schaefer. Notre théorème principal détaille cette situation, énonçant un nouveau résultat dichotomique pour l'énumération des modèles par poids croissant.

Théorème 2.2 *(*Enumération par poids croissant*.) Si* Γ *est* $Horn$ *ou* $Krom$*, alors il existe un algorithme à délai polynomial qui génère tous les modèles d'une* Γ*-formule par poids croissant. Sinon, un tel algorithme n'existe pas, à moins que* P = NP.

Ce résultat est prouvé dans les deux prochaines sections.

3 Algorithmes d'énumération efficaces

Les algorithmes d'énumération efficaces proposés dans [4] (voir Théorème 2.1) sont implicitement basés sur l'auto-réductibilité ([23, 13, 20]). Par conséquent ces algorithmes énumèrent les modèles dans l'ordre lexicographique et n'utilisent qu'un espace polynomial. Enumérer les solutions dans l'ordre du poids croissant exige de nouveaux algorithmes. Les deux classes de langages de contrainte examinées dans cette section, $Krom$ et $Horn$, demandent des approches algorithmiques différentes. En effet elles se distinguent par la complexité de leur problème associé k-Ones, dans lequel, étant donné une formule et un entier k, on veut savoir s'il existe un modèle avec exactement k uns. Pour des formules $Krom$ ce problème est dans P, alors que pour des formules de $Horn$ il est NP-complet (voir [6]). En conséquence, dans le cas des formules $Krom$ on peut exploiter le fait que k-Ones est résoluble en temps polynomial pour obtenir un algorithme d'énumération efficace. En revanche, les formules de $Horn$ demandent une autre stratégie. Il est naturel d'utiliser une structure de données qui maintient un ensemble ordonné et qui permet des opérations efficaces d'insertion et d'extraction. Ainsi, l'algorithme que nous allons présenter pour le cas $Horn$ utilise une file de priorité. Cette méthode a déjà été utilisée dans e.g. [10, 14]. Comme dans ces articles, notre file de priorité permet l'insertion et l'extraction de l'élément de tête en temps logarithmique en la taille de l'entrée. Ainsi la taille de la file peut augmenter de façon exponentielle tandis qu'un délai polynomial est toujours maintenu.

Proposition 3.1 *Si* Γ *est Krom, alors il existe un algorithme à délai polynomial et en espace polynomial qui énumère tous les modèles d'une* Γ*-formule par poids croissant.*

Preuve : Soient Γ un ensemble de relations Krom et φ une Γ-formule. On peut supposer sans perte de généralité que φ ne contient pas de clauses unitaires. Ainsi chaque clause de φ exprime soit l'égalité, soit la non-égalité entre deux variables. En utilisant la transitivité de l'égalité et le fait que dans le cas booléen $a \neq b \neq c$ implique $a = c$, on peut identifier des classes d'équivalence de variables telles que deux classes quelconques ou bien sont indépendantes, ou bien correspondent à des valeurs de vérité opposées. On appelle *cluster* une paire (A, B) de classes correspondant à des valeurs de vérité opposées, B pouvant être vide. Il est immédiat que deux clusters quelconques sont nécessairement indépendants l'un de l'autre et donc afin d'obtenir un modèle de φ, il suffit de choisir pour chaque cluster (A, B) ou bien $A = 1, B = 0$ ou bien $A = 0, B = 1$. Dans la suite nous supposons que φ est satisfaisable (dans le cas contraire nous détecterons une contradiction en contruisant les clusters). Soit $n \geq 1$ le nombre de clusters, alors le nombre de modèles est 2^n. La contribution de chaque cluster en terme de poids est ou bien $|A|$ ou bien $|B|$ (il peut également arriver que $|A| = |B|$). Représentons un modèle par un n-uplet $s \in \{0, 1\}^n$ indiquant pour chaque cluster l'affectation qui est choisie. Dans le cas où $|A| \neq |B|$ nous indiquons

par 0 l'affectation de plus faible poids et par 1 l'autre. Ainsi $(0, 0, \ldots, 0)$ représentera un modèle de poids minimal, alors que $(1, 1, \ldots, 1)$ représentera un modèle de poids maximal. Pour l'énumération nous considérons seulement la différence de poids $\big||A| - |B|\big|$ de chaque cluster puisque nous pouvons au final soustraire le poids d'un modèle minimal. En posant $(w_1, \ldots, w_n)$ pour représenter les différences de poids de chacun des clusters nous sommes finalement ramenés au problème d'énumération suivant :

Problème :	UNARY-SUBSET-SUM
Entrée :	Une suite d'entiers positifs ou nuls $(w_1, \ldots, w_n) \in \mathbb{N}^n$ codés en unaire
Sortie :	énumérer tous les n-uplets $s \in \{0,1\}^n$ par poids $\delta(s)$ croissant, où $\delta(s) = \Sigma_{i=1}^{n} s_i \cdot w_i$

Observons que la taille de l'entrée est donnée par $\max(n, W)$ où $W := \Sigma_{i=1}^{n} w_i$ puisque les entiers sont codés en unaire. De plus la réduction de notre problème initial au problème UNARY-SUBSET-SUM est bien calculable en temps polynomial puisque la somme totale des poids W est majorée par le nombre de variables de la formule intiale (et donc la représentation des entiers en unaire n'est pas un obstacle). Pour achever la preuve de la proposition il nous reste à montrer que le problème UNARY-SUBSET-SUM est résoluble avec un délai polynomial. Pour ce faire nous utilisons une stratégie de programmation dynamique pour calculer en temps polynomial une matrice $A \in \{0,1\}^{(n+1,\, W+1)}$ telle que $A(i,k) = 1$ si et seulement si en sommant certains des poids $w_1, \ldots, w_i$ on peut atteindre la somme k, où $0 \leq i \leq n,\ 0 \leq k \leq W$. La matrice A est construite en posant tout d'abord $A(0,0) = 1$ et $A(0,k) = 0$ pour tout $k \geq 1$, puis en remplissant les autres champs ligne par ligne selon la règle $A(i,k) = 1$ si et seulement si $A(i-1,k) = 1$ ou $A(i-1, k-w_i) = 1$. Ainsi le calcul de A requiert un temps $O(n \cdot W)$. Ce pré-calcul étant fait, pour chaque poids k d'une solution, nous énumérons toutes ces solutions de poids k. Ceci se fait récursivement en construisant chaque solution, représentée par un mot binaire, à partir du mot vide (ϵ).

Le lecteur pourra se convaincre facilement que l'Algorithme 1 énumère toutes les solutions s du problème UNARY-SUBSET-SUM par poids $\delta(s)$ croissant. Le précalcul demande un temps quadratique et le délai est linéaire ensuite. Au final nous obtenons donc bien un algorithme d'énumération à délai polynomial qui utilise un espace quadratique pour l'énumération des modèles d'une formule Krom par poids croissant. □

Algorithm 1 Algorithme pour UNARY-SUBSET-SUM.

```
MAIN(w_1, ..., w_n)
 1: compute A ∈ {0,1}^(n+1, W+1)
 2: for k = 0 to W do
 3:    if A(n,k) = 1 then
 4:       CONSTRUCTSOLUTIONS(n, k, ε)
            /* énumération des solutions de poids k */
 5: end for

CONSTRUCTSOLUTIONS(i, j, s)
 1: if i = 0 then
 2:    output s
 3: else
 4:    if A(i-1, j-w_i) = 1 then
 5:       CONSTRUCTSOLUTIONS(i-1, j - w_i, 1 ∘ s)
          /* ∘ est l'opérateur de concaténation */
 6:    if A(i-1, j) = 1 then
 7:       CONSTRUCTSOLUTIONS(i-1, j, 0 ∘ s)
```

Proposition 3.2 *Si* Γ *est Horn, alors il existe un algorithme à délai polynomial qui énumère tous les modèles d'une* Γ*-formule par poids croissant.*

Preuve : Soit φ une Γ-formule. Si Γ est *Horn* alors φ est équivalente à une conjonction de clauses de *Horn*. Nous utilisons une file de priorité Q pour respecter l'ordre par poids croissant et pour éviter des doublons. La commande Q.enqueue(s, k) insère un élément s avec une clé k (un poids). La file trie par clé croissante et insère un élément s seulement s'il n'est pas encore présent dans la file.

Pour plus de lisibilité nous représentons un modèle par l'ensemble de variables assignées à 1. Nous utilisons le fait bien connu que l'intersection de tous les modèles d'une formule de *Horn* est l'unique *modèle minimal* qui peut être calculé en temps polynomial. Pour φ une formule de *Horn* satisfaisable nous indiquons le modèle minimal par $mm(\varphi)$. Remarquons que pour un ensemble de variables $V \subseteq \mathrm{Var}(\varphi)$ la formule $\varphi \wedge V := \varphi \wedge \bigwedge_{v \in V} v$ est encore une formule de *Horn*.

Nous affirmons qu'étant donné une formule de *Horn*, l'Algorithme 2 énumère tous ses modèles par poids croissant avec délai polynomial. Le fait que le délai soit polynomial est immédiat. Selon la définition de la file de priorité, et puisque les modèles m' générés à partir de m en ligne 11 sont toujours de poids supérieur à celui de m, il est également facile de vérifier que les modèles sont générés dans le bon ordre et qu'aucun modèle n'est généré deux fois. Pour prouver qu'aucun modèle n'est omis, il suffit de démontrer que pour tout modèle $m' \neq mm(\varphi)$ il existe un sous-modèle $m \subsetneq m'$ tel qu'en ligne 11 l'algorithme génère m' à partir de m. C'est-à-dire il faut qu'il y ait un $x \in m' \setminus m$ tel que $m' = mm(\varphi \wedge m \wedge x)$. Pour cela nous considérons l'ensemble $H := \{m \mid m \text{ un modèle de } \varphi \text{ et } m \subsetneq m'\}$.

Algorithm 2 Algorithm pour HORN-SAT.

Require: φ a Horn formula
1: **if** φ unsatisfiable **then**
2: return 'no'
3: Q = newPriorityQueue
4: $m := mm(\varphi)$
5: Q.enqueue(m, $|m|$)
6: **while** Q not empty **do**
7: m := Q.dequeue
8: output m
9: **for all** $x \in \mathrm{Var}(\varphi) \setminus m$ **do**
10: **if** $\varphi \wedge m \wedge x$ satisfiable **then**
11: $m' := mm(\varphi \wedge m \wedge x)$
12: Q.enqueue(m', $|m'|$)
13: **end for**
14: **end while**

L'ensemble H n'est pas vide car il contient au moins le modèle minimal $mm(\varphi)$. Un modèle maximal m de H satisfait toutes les conditions requises car il vérifie $m' = mm(\varphi \wedge m \wedge x)$ pour tout $x \in m' \setminus m$.

Finalement soulignons que contrairement à l'Algorithme 1, l'Algorithme 2 utilise potentiellement un espace exponentiel. □

4 Cas difficiles

Dans cette section nous nous intéressons au cas où Γ n'est ni Horn ni Krom. Clairement pour pouvoir énumérer tous les modèles d'une Γ-formule par poids croissant il est nécessaire d'être capable de trouver un modèle de poids minimal en temps polynomial. Comme nous allons le voir ce n'est pas une condition suffisante. En effet il faut également être capable de trouver le deuxième modèle efficacement. Les problèmes suivants vont jouer un rôle important dans notre étude :

Problème :	MIN-ONES(Γ)
Entrée :	une Γ-formule φ, un entier positif W
Question:	Existe-t-il un modèle de φ qui attribue la valeur vrai à au plus W variables ?
Problème :	MIN-ONES*(Γ)
Entrée :	une Γ-formule φ, un entier positif W
Question:	Existe-t-il un modèle de φ non identiquement nul qui attribue la valeur vrai à au plus W variables ?

De la classification obtenue dans [12] pour le problème d'optimisation correspondant on peut déduire ce qui suit.

Proposition 4.1 *Si* Γ *est* 0*-valide ou Horn ou Krom, alors* MIN-ONES(Γ) *est dans* P*, sinon* MIN-ONES(Γ) *est* NP*-complet.*

Notre principale contribution dans cette partie est le résultat de complétude suivant. Il prouve de façon évidente que si Γ n'est ni Horn ni Krom, alors il n'existe pas d'algorithme qui énumère tous les modèles d'une Γ-formule par poids croissant avec un délai polynomial à moins que P = NP.

Proposition 4.2 *Soit* Γ *un ensemble de relations qui n'est ni Horn ni Krom. Alors* MIN-ONES*(Γ) *est* NP*-complet.*

Preuve : Si Γ n'est pas Schaefer, alors SAT*(Γ) est NP-complet [19] et donc *a fortiori* le problem MIN-ONES*(Γ) l'est également. Si Γ n'est pas 0-valide, alors, puisqu'il n'est ni Horn ni Krom, le résultat découle de la NP-complétude de MIN-ONES(Γ) (Proposition 4.1). Ainsi, il reste à étudier les ensembles Γ qui sont Schaefer et 0-valides mais qui ne sont ni Horn ni Krom. Il y a trois cas à analyser.

- Γ est bijonctif et 0-valide mais ni Horn ni Krom.
- Γ est affine et 0-valide mais ni Horn ni Krom.
- Γ est dual-Horn et 0-valide mais ni Horn ni Krom.

Observer qu'une formule sous forme normale conjonctive avec deux littéraux dans chaque clause qui est 0-valide est également Horn. Ainsi le premier cas n'arrive jamais. De plus on peut facilement prouver qu'une relation 0-valide et affine qui n'est pas Horn ne peut-être Krom. Pour cette raison, afin d'établir la preuve de la proposition il est suffisant de prouver successivement la NP-complétude de MIN-ONES*(Γ) dans les deux cas suivants :

1. Γ est affine et 0-valide mais pas Horn, ou
2. Γ est dual-Horn et 0-valide mais ni affine ni Horn.

La NP-complétude de MIN-ONES*(Γ) pour tout ensemble Γ vérifiant une de ces deux conditions est établie respectivement dans les Propositions 4.6 et 4.9. Le schéma de la preuve donné par la Figure 1 convaincra facilement le lecteur qu'au final aucun cas n'a été oublié. □

4.1 Affine, 0-valide, non Horn

Dans cette partie nous traitons les relations qui sont 0-valides et affines mais non Horn. Nous allons prouver que pour une telle relation R, trouver un modèle non identiquement nul de poids minimal d'une R-formule est NP-difficile. Pour ce faire nous avons besoin de quelques lemmes techniques. Les deux premiers sont des résultats de définissabilité, alors que le troisième

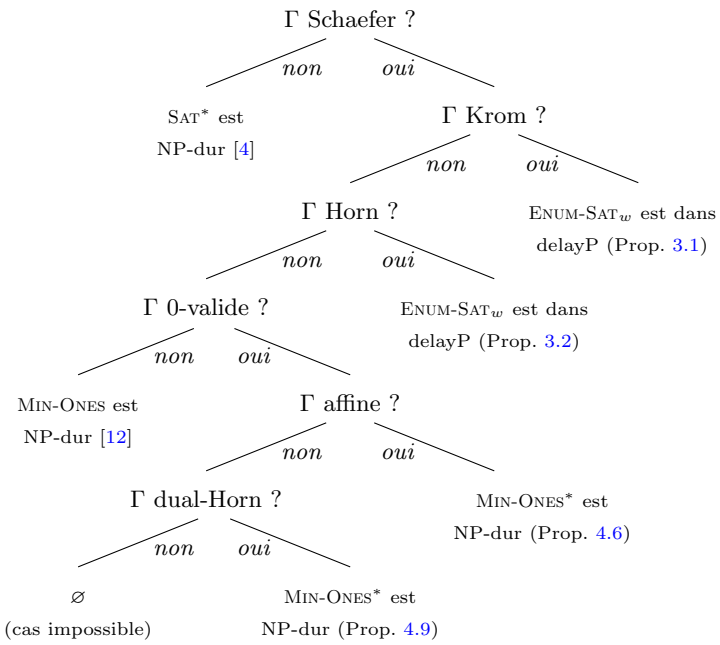


FIGURE 1 – Schéma de la preuve

est un résultat de complétude pour un problème de base.

Une des méthodes les plus populaires pour obtenir des résultats de complexité pour des problèmes liés à la satisfaction de contraintes (notamment pour l'énumération) est d'utiliser des outils issus de l'algèbre universelle. Une correspondance de Galois établit en effet un lien entre le pouvoir d'expressibilité d'un langage de contraintes et son ensemble de polymorphismes ou polymorphismes partiels (voir e.g. [9, 22]). Cependant dans le cas présent il n'est pas nécessaire d'utiliser ces outils très élaborés. En effet les résultats techniques dont nous avons besoin concernent des ensembles de relations très particuliers et peuvent être obtenus à la main.

Dans les preuves de tous les lemmes qui suivent R représente une relation d'arité k et V un ensemble de k variables deux à deux distinctes, mettons $V = \{x_1, \ldots, x_k\}$.

Lemme 4.3 *Soit R une relation qui est 0-valide et affine mais ni Horn ni 1-valide. Alors il existe une R-formule équivalente à $\neg w \wedge (x \oplus y \oplus z = 0)$.*

Preuve : Considérons la contrainte $C = R(x_1, \ldots, x_k)$. Puisque R n'est pas Horn il existe m_1 et m_2 dans R tels que $m_1 \wedge m_2 \notin R$. Etant donné que R est 0-valide et affine nous avons $m_1 \oplus m_2 \in R$. Pour $i, j \in \{0, 1\}$, soit $V_{i,j} = \{x \mid x \in V, m_1(x) = i \wedge m_2(x) = j\}$. Observons que $V_{0,1} \neq \varnothing$ (respectivement, $V_{1,0} \neq \varnothing$), sinon $m_1 \wedge m_2 = m_2$ (resp., $m_1 \wedge m_2 = m_1$), ce qui contredit le fait que $m_1 \wedge m_2 \notin R$. De plus $V_{1,1} \neq \varnothing$, sinon $m_1 \wedge m_2 = \vec{0}$, une contradiction. Considérons la $\{R\}$-contrainte : $M(w, x, y, z) = C[w/V_{0,0}, x/V_{0,1}, y/V_{1,0}, z/V_{1,1}]$. Selon la remarque ci-dessus les trois variables x, y and z apparaissent effectivement dans cette contrainte. Examinons l'ensemble des modèles de M qui attribuent la valeur 0 à w : il contient 0011 (car $m_1 \in R$), 0101 (car $m_2 \in R$), 0110 (car $m_1 \oplus m_2 \in R$) et 0000 (car R est 0-valide). Cependant il ne contient pas 0001 (car par hypothèse $m_1 \wedge m_2 \notin R$). Il ne contient pas non plus 0111. En effet, sinon il contiendrait $0011 \oplus 0101 \oplus 0111$ (car R est affine), qui est équivalent à 0001, contradiction. On peut également déduire que cet ensemble ne contient pas non plus 0010 ni 0100 (car $0000 \oplus 0011 \oplus 0010 = 0001$ et $0110 \oplus 0101 \oplus 0100 = 0111$). Remarquons que puisque R est 0-valide mais non-1-valide $C[w/V] \equiv \neg w$. Considérons alors

$$\varphi(w, x, y, z) = C[w/V] \wedge M(w, x, y, z).$$

Cette R-formule φ est équivalente à $\neg w \wedge (x \oplus y \oplus z = 0)$, ce qui termine la preuve. □

Lemme 4.4 *Soit R une relation qui est 0-valide, 1-valide, affine mais pas Horn. Alors il existe une R-formule équivalente à $(w \oplus x \oplus y \oplus z = 0)$.*

Preuve : Observons qu'une relation R qui est affine et à la fois 0-valide et 1-valide est nécessairement complémentive, i.e., pour tout $m \in R$ nous avons également $\vec{1} \oplus m \in R$. Nous pouvons imiter l'analyse faite dans le lemme précédent et considérer $M(w, x, y, z) = C[w/V_{0,0}, x/V_{0,1}, y/V_{1,0}, z/V_{1,1}]$. Ainsi $\varphi(w, x, y, z) = M(w, x, y, z)$ vérifie $\varphi(w, x, y, z) \equiv (w \oplus x \oplus y \oplus z = 0)$. □

Lemme 4.5 MIN-ONES*$(x \oplus y \oplus z = 0)$ *et* MIN-ONES*$(w \oplus x \oplus y \oplus z = 0)$ *sont NP-complets.*

Preuve : Soit S un système linéaire homogène dans le corps fini à deux éléments GF(2). Il est connu que trouver une solution non identiquement nulle de poids minimal d'un tel système est NP-difficile (voir [2, Théorème 4.1]). Afin de prouver le lemme ci-dessus nous devons prouver que ce problème reste difficile même quand il est restreint à des systèmes dont chaque équation a seulement trois (resp., quatre) variables. Supposons que S a n variables, $x_1, \ldots, x_n$. Afin de réduire le nombre de variables par équation nous introduisons des variables auxiliaires. S'il existe une équation $x_{i_1} \oplus x_{i_2} \oplus \cdots \oplus x_{i_k} = 0$ pour un certain $k \geq 4$,

nous introduisons une nouvelle variable, y_{i_1,i_2}, et remplaçons l'équation originale par les deux équations $y_{i_1,i_2} \oplus x_{i_1} \oplus x_{i_2} = 0$ et $y_{i_1,i_2} \oplus x_{i_3} \oplus \ldots \oplus x_{i_k} = 0$. Nous réitérons ce procédé jusqu'à ce que toutes les équations aient trois variables. La satisfaisabilité du système est préservée durant cette transformation. Le nombre de variables auxiliaires est majoré par le nombre d'occurrences de variables dans le système initial. Afin de conserver l'information sur le poids des solutions l'idée est d'introduire suffisamment de copies des variables originales, rendant ainsi les variables auxiliaires négligables. Soit N le nombre d'occurrences de variables dans S. Soit f une nouvelle variable qui va jouer le rôle de la constante 0. Pour tout $i = 1, \ldots, n$, nous introduisons N copies de x_i, $x_i^1, \ldots, x_i^N$, et ajoutons les équations $x_i \oplus x_i^j \oplus f = 0$ pour $j = 1, \ldots N$. Finalement nous ajoutons l'équation $f \oplus f \oplus f = 0$, i.e., $f = 0$ (cette équation va assurer que $x_i = x_i^j$ pour tout j). Il y a une correspondance bijective entre les solutions de S et les solutions du système S' ainsi obtenu. De plus S a une solution non triviale de poids au plus W si et seulement si S' a une solution non triviale de poids au plus $W(N+1)+N$. Puisque le système S' peut-être vu comme une $(x \oplus y \oplus z = 0)$-formule, nous avons ainsi prouvé la NP-difficulté de Min-Ones$^*(x \oplus y \oplus z = 0)$.

Réduisons maintenant Min-Ones$^*(x \oplus y \oplus z = 0)$ à Min-Ones$^*(w \oplus x \oplus y \oplus z = 0)$. Soit S un système linéaire homogène à n variables tel que chaque équation comporte exactement trois variables. Soient w et w_i pour $i = 1, \ldots, n+1$ de nouvelles variables. Transformons S en S' comme suit : transformons chaque équation $x \oplus y \oplus z = 0$ en $w \oplus x \oplus y \oplus z = 0$ et ajoutons les $n+1$ equations $w \oplus w \oplus w \oplus w_i = 0$ pour $i = 1, \ldots, n+1$. Les solutions de S' affectant la valeur 0 à w coïncident avec les solutions de S. De plus toute solution de S' affectant la valeur 1 à w a un poids au moins égal à $n+1$. Donc, S a une solution non triviale de poids au plus W ($W \leq n$) si et seulement si S' a une solution non triviale de poids au plus W, c.q.f.d. □

Nous pouvons désormais énoncer notre résultat de difficulté.

Proposition 4.6 *Si R est 0-valide et affine mais non Horn, alors le problème* Min-Ones$^*(R)$ *est* NP*-complet.*

Preuve : La proposition découle des trois lemmes ci-dessus : si R n'est pas 1-valide, alors selon le Lemme 4.3 on peut réduire le problème Min-Ones$^*(x \oplus y \oplus z = 0)$ au problème Min-Ones$^*(R)$ (remplacer chaque contrainte $(x \oplus y \oplus z = 0)$ par la R-formule équivalente à $\neg w \wedge (x \oplus y \oplus z = 0)$, où w est une nouvelle variable). Si la relation R est 1-valide, alors selon le Lemme 4.4 on peut réduire Min-Ones$^*(w \oplus x \oplus y \oplus z = 0)$ à Min-Ones$^*(R)$. Dans les deux cas le Lemma 4.5 permet de conclure. □

4.2 Dual-Horn, 0-valide, ni Krom ni Horn

Dans cette partie nous nous intéressons aux relations qui sont 0-valides et dual-Horn mais ni affines ni Horn. La méthode de preuve n'est pas la même que celle de la section précedente. Néanmoins nous aurons également besoin de quelques résultats techniques intermédiaires.

Lemme 4.7 *Soit R une relation qui est 0-valide et dual-Horn mais ni affine ni 1-valide. alors il existe une R-formule équivalente à $\neg t \wedge (u \rightarrow v)$.*

Preuve : Considérons la contrainte $C = R(x_1, \ldots, x_k)$. Observons que puisque R est 0-valide mais non 1-valide, $C[t/V] \equiv \neg t$. Puisque R est 0-valide mais pas affine il existe deux tuples distincts m_1 et m_2 dans R tels que $m_1 \oplus m_2 \notin R$. Puisque R est dual-Horn, nous avons $m_1 \vee m_2 \in R$. Pour $i, j \in \{0, 1\}$, soit $V_{i,j} = \{x \mid x \in V, m_1(x) = i \wedge m_2(x) = j\}$. Observons que $V_{1,1} \neq \varnothing$, sinon $m_1 \vee m_2 = m_1 \oplus m_2$, une contradiction. De plus, puisque $m_1 \neq m_2$ ou bien $V_{0,1}$ ou $V_{1,0}$ est non vide. Supposons dans un premier temps qu'ils sont tous les deux non vides. Considérons la R-contrainte $M(w, x, y, z) = C[w/V_{0,0}, x/V_{0,1}, y/V_{1,0}, z/V_{1,1}]$. Les trois variables x, y and z apparaissent effectivement dans cette contrainte. Examinons l'ensemble des modèles de M qui affectent la valeur 0 à w. Cet ensemble contient 0011 (car $m_1 \in R$), 0101 (car $m_2 \in R$), 0111 (car $m_1 \vee m_2 \in R$) et 0000 (car R est 0-valide), mais ne contient pas 0110 (puisque par hypothèse $m_1 \oplus m_2 \notin R$). L'appartenance de 0100, 0010, 0001 à cet ensemble est ouverte :

- S'il ne contient pas 0100, alors considérons la R-formule $\varphi(t, u, v) := C[t/V] \wedge M(t, u, v, v)$. Son ensemble de modèles est $\{001, 011, 000\}$, et donc $\varphi(t, u, v) \equiv \neg t \wedge (u \rightarrow v)$.
- S'il contient 0100, alors il ne contient pas 0010. En effet sinon, puisque R est dual-Horn, il contiendrait également 0110, ce qui fournit une contradiction. Considérons donc la R-formule $\varphi(t, u, v) := C[t/V] \wedge M(t, v, u, v)$. Son ensemble de modèles est $\{001, 011, 000\}$, et donc $\varphi(t, u, v) \equiv \neg t \wedge (u \rightarrow v)$.

Si par exemple $V_{0,1} = \varnothing$, alors on doit considérer $M(w, y, z) = C[w/V_{0,0}, y/V_{1,0}, z/V_{1,1}]$. Dans ce cas $\varphi(t, u, v) := C[t/V] \wedge M(t, u, v)$ est équivalent à $\neg t \wedge (u \rightarrow v)$. □

Lemme 4.8 *Soit R une relation qui est 0-valide, 1-valide, et dual-Horn mais pas Horn, alors il existe une R-formule équivalente à $(u \rightarrow v)$.*

Preuve : Puisque R est dual-Horn mais pas Horn, elle n'est pas complémentive, c'est-à-dire il existe un certain tuple $m \in R$ tel que $m \oplus \vec{1} \notin R$. Considérons alors la contrainte $C = R(x_1, \ldots, x_k)$. Pour $i \in \{0, 1\}$, soit $V_i = \{x \mid x \in V \wedge m(x) = i\}$. Considérons la R-formule $\varphi(u, v) = C[u/V_0, v/V_1]$. Alors $\varphi(u, v) \equiv \neg u \vee v \equiv u \rightarrow v$. □

Proposition 4.9 *Si R est 0-valide et dual-Horn mais n'est ni affine ni Horn, alors le problème* Min-Ones$^*(R)$ *est* NP*-complet.*

Preuve : Soit R une telle relation qui est 0-valide et dual-Horn mais n'est ni affine ni Horn. Soit T la relation unaire constante, $T = \{1\}$. Selon la Proposition 4.1, le problème Min-Ones(R, T) est NP-complet. Réduisons Min-Ones(R, T) à Min-Ones$^*(R)$. Soit φ une $\{R, T\}$-formule, $\varphi = \psi \wedge \bigwedge_{x \in V} T(x)$ où ψ est une R-formule. Soit t une nouvelle variable. Considérons

$$\varphi' = \psi[t/V] \wedge \bigwedge_{x \in \mathrm{Var}(\varphi) \setminus V} x \rightarrow t.$$

Observons que la seule solution qui affecte la valeur 0 à t dans φ' est la solution identiquement nulle. Pour cette raison il est clair que φ a une solution de poids au plus W ($W \geq |V|$) si et seulement si φ' admet une solution non identiquement nulle de poids au plus $W - \mid V \mid +1$. Les deux lemmes ci-dessus permettent d'exprimer φ' comme une R-formule (modulo l'introduction variable supplémentaire qui va toujours prendre la valeur 0 quand R n'est pas 1-valide), la preuve est ainsi complète. □

5 Conclusion

Nous avons classifié la complexité de l'énumération des modèles d'une Γ-formule par poids croissant selon Γ. Nous avons prouvé que dans le cas des problèmes de satisfaction de contraintes booléennes une condition nécessaire et suffisante pour énumérer toutes les solutions par poids croissant avec un délai polynomial est le fait d'être capable de fournir une solution non identiquement nulle de poids minimal en temps polynomial. Remarquons que par dualité, sous l'hypothèse P ≠ NP, on peut énumérer tous les modèles d'une Γ-formule par poids *décroissant* avec un délai polynomial si et seulement si Γ est Krom ou dual-Horn. Nous aurions également pu considérer la version pondérée du problème, dans laquelle il y a une fonction de poids $w\colon V \rightarrow \mathbb{N}$ et le poids d'un modèle m est donné par $\sum_{v \in V} w(v) \cdot m(v)$. L'algorithme proposé dans la preuve de la Proposition 3.2 pour les formules de Horn pourrait tout aussi bien traiter cette variante. En revanche tel n'est pas le cas de l'algorithme proposé pour les formules Krom (Proposition 3.1). En effet cet algorithme est basé sur une procédure de programmation dynamique qui est pseudo-polynomiale et qui devient de complexité exponentielle quand les poids ne sont plus polynomialement bornés. Cependant, au prix d'un espace exponentiel, on peut développer un algorithme d'énumération à délai polynomial pour les formules Krom en utilisant une file de priorité comme cela a été proposé dans le cas des formules de Horn. Le fait d'exiger que l'énumération soit faite par poids croissant a révélé de nouveaux algorithmes d'énumération pour les problèmes de satisfaction de contraintes booléennes, différents de ceux proposés jusque-là. L'algorithme développé pour les formules de Horn requiert potentiellement un espace exponentiel, il serait souhaitable bien sûr d'éviter cela. Il serait intéressant de savoir s'il est possible de réduire cet espace utilisé au prix peut-être d'une augmentation de la complexité en temps ou à l'inverse si cette quantité d'espace exigée est inhérente à cet ordre particulier (par poids croissant) pour les formules de Horn. Une autre piste de recherche est l'étude de l'énumération par poids croissant des solutions de problèmes de satisfaction de contraintes sur des domaines finis de cardinalité supérieure à 2. L'énumération des solutions de tels problèmes a déjà été étudiée dans [3, 21]. Comme mentionné dans [21] le fait d'exiger un ordre pré-défini dans l'énumération pourrait être la clé pour découvrir de nouveaux algorithmes. Dans [11] les auteurs ont étudié la complexité du problème Max-Sol(Γ) qui généralise le problème Max-Ones à des domaines finis arbitraires. Ils ont identifié deux classes de langages de contraintes pour lesquelles ce problème peut-être résolu efficacement. Ces deux classes peuvent être vues comme des généralisations substantielles et non triviales des classes pour lesquelles le problème Max-Ones est efficacement résoluble sur le domaine booléen, c'est-à-dire, Krom, 1-valide et dual-Horn. Il serait intéressant d'examiner ces classes plus en détail afin de déterminer si elles peuvent donner lieu à des algorithmes à délai polynomial pour l'énumération des solutions par poids croissant.

Remerciements

Nous remercions Arnaud Durand pour nous avoir fourni une précieuse référence, [2].

Références

[1] G. Bagan, A. Durand, and E. Grandjean. On acyclic conjunctive queries and constant delay enumeration. In J. Duparc and T. A. Henzinger, edi-

tors, *CSL*, volume 4646 of *LNCS*, pages 208–222. Springer, 2007.

[2] A. Barg. Complexity issues in coding theory. *Electronic Colloquium on Computational Complexity (ECCC)*, 4(46), 1997.

[3] D. A. Cohen. Tractable decision for a constraint language implies tractable search. *Constraints*, 9(3) :219–229, 2004.

[4] N. Creignou and J.-J. Hébrard. On generating all solutions of generalized satisfiability problems. *Theoretical Informatics and Applications*, 31(6) :499–511, 1997.

[5] N. Creignou, Ph. G. Kolaitis, and H. Vollmer, editors. *Complexity of Constraints - An Overview of Current Research Themes*, volume 5250 of *LNCS*. Springer, 2008.

[6] N. Creignou, H. Schnoor, and I. Schnoor. Nonuniform boolean constraint satisfaction problems with cardinality constraint. *ACM Trans. Comput. Log.*, 11(4), 2010.

[7] N. Creignou and H. Vollmer. Boolean constraint satisfaction problems : When does Post's lattice help ? In Creignou et al. [5], pages 3–37.

[8] M. Hagen. Lower bounds for three algorithms for transversal hypergraph generation. *Discrete Applied Mathematics*, 2009.

[9] P. G. Jeavons. On the algebraic structure of combinatorial problems. *Theoretical Computer Science*, 200 :185–204, 1998.

[10] D. S. Johnson, C. H. Papadimitriou, and M. Yannakakis. On generating all maximal independent sets. *Inf. Process. Lett.*, 27(3) :119–123, 1988.

[11] P. Jonsson and G. Nordh. Introduction to the maximum solution problem. In Creignou et al. [5], pages 255–282.

[12] S. Khanna, M. Sudan, and D. Williamson. A complete classification of the approximability of maximization problems derived from Boolean constraint satisfaction. In *Proceedings 29th Symposium on Theory of Computing*, pages 11–20. ACM Press, 1997.

[13] S. Khuller and V. V. Vazirani. Planar graph coloring is not self-reducible, assuming P $\neq$ NP. *Theoretical Computer Science*, 88(1) :183–189, 1991.

[14] B. Kimelfeld and Y. Sagiv. Incrementally computing ordered answers of acyclic conjunctive queries. In O. Etzion, T. Kuflik, and A. Motro, editors, *NGITS*, volume 4032 of *LNCS*, pages 141–152. Springer, 2006.

[15] B. Kimelfeld and Y. Sagiv. Efficiently enumerating results of keyword search over data graphs. *Inf. Syst.*, 33(4-5) :335–359, 2008.

[16] A. A. Krokhin and D. Marx. On the hardness of losing weight. In L. Aceto, I. Damgård, L. A. Goldberg, M. M. Halldórsson, A. Ingólfsdóttir, and I. Walukiewicz, editors, *ICALP (1)*, volume 5125 of *LNCS*, pages 662–673. Springer, 2008.

[17] K. Makino and T. Uno. New algorithms for enumerating all maximal cliques. In *Scandinavian Workshop on Algorithm Theory*, pages 260–272, 2004.

[18] D. Marx. Parameterized complexity of constraint satisfaction problems. *Computational Complexity*, 14(2) :153–183, 2005.

[19] T. J. Schaefer. The complexity of satisfiability problems. In *Proccedings 10th Symposium on Theory of Computing*, pages 216–226. ACM Press, 1978.

[20] J. Schmidt. Enumeration : Algorithms and complexity. Preprint (2009), available at http://www.thi.uni-hannover.de/fileadmin/forschung/arbeiten/schmidt-da.pdf.

[21] H. Schnoor and I. Schnoor. Enumerating all solutions for constraint satisfaction problems. In W. Thomas and P. Weil, editors, *STACS*, volume 4393 of *LNCS*, pages 694–705. Springer, 2007.

[22] H. Schnoor and I. Schnoor. Partial polymorphisms and constraint satisfaction problems. In Creignou et al. [5], pages 229–254.

[23] C. P. Schnorr. Optimal algorithms for self-reducible problems. In *International Conference on Automata, Languages and Programming*, pages 322–337, 1976.

[24] Y. Strozecki. Enumeration complexity and matroid decomposition. *Phd thesis*, 2010.

[25] V. Vazirani and M. Yannakakis. Suboptimal cuts : Their enumeration, weight and number. In W. Kuich, editor, *Automata, Languages and Programming*, volume 623 of *Lecture Notes in Computer Science*, pages 366–377. Springer Berlin / Heidelberg, 1992.

[26] L.-P. Yeh, B.-F. Wang, and H.-H. Su. Efficient algorithms for the problems of enumerating cuts by non-decreasing weights. *Algorithmica*, 56(3) :297–312, 2010.

Algorithmes de filtrage pour des problèmes cumulatifs discrets avec dépassements de ressource

Alexis De Clercq Thierry Petit Nicolas Beldiceanu Narendra Jussien

Mines-Nantes, LINA UMR CNRS 6241,
4, rue Alfred Kastler, FR-44307 Nantes, France.
{Alexis.De-Clercq, Thierry.Petit, Nicolas.Beldiceanu, Narendra.Jussien}@mines-nantes.fr

Abstract

Dans de nombreux problèmes cumulatifs, l'horizon est fixé et ne peut être retardé. Dans cette situation, il arrive souvent que toutes les activités ne puissent pas être ordonnancées sans dépasser la capacité en certains points de temps. De plus, cette capacité n'est pas nécessairement la même sur toute la fenêtre temporelle. Dans ce contexte, cet article introduit une nouvelle contrainte pour résoudre cette classe de problèmes. Nous adaptons deux algorithmes à notre contexte : Sweep et l'algorithme d'Edge-Finding de P. Vilìm. Comme dans certains problèmes des dépassements minimaux sont imposés, nous décrivons une procédure spécifique à ce type d'événements. Nous introduisons une heuristique de recherche spécifique à notre contrainte. Nous expérimentons avec succès notre contrainte.

1 Introduction

Les problèmes d'ordonnancement consistent à ordonner des activités. En *ordonnancement cumulatif*, chaque activité a une durée et nécessite, pour son exécution, la disponibilité d'une certaine quantité de ressource renouvelable, sa *consommation* (ou *demande en capacité*). Habituellement, l'objectif est de minimiser *l'horizon* (la date de fin maximale d'une activité), tandis qu'en chaque point de temps la consommation cumulée des activités ne doit pas excéder une limite sur la ressource disponible, la *capacité*. Cependant, de nombreux problèmes industriels nécessitent que les activités soient ordonnancées dans une fenêtre temporelle donnée : l'horizon est fixé et ne peut être retardé. Dans cette situation, il peut arriver que toutes les activités ne puissent pas être ordonnancées sans dépasser la capacité en certains points de temps. Pour obtenir une solution, les dépassements peuvent être tolérés dans une certaine limite, à condition que des règles opérationnelles garantissant la faisabilité pratique de la solution soient satisfaites. De plus, dans certains problèmes, la fenêtre temporelle est partitionnée : la capacité n'est pas la même pour chaque intervalle de la partition. La Figure 1 illustre une telle situation avec des dépassements.

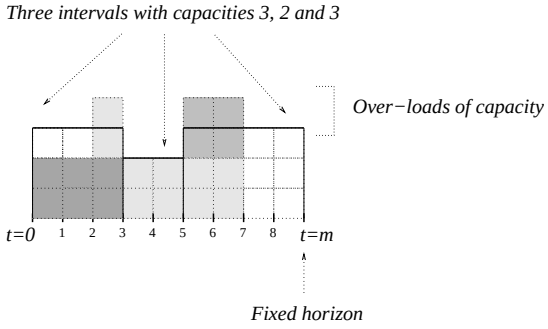


FIGURE 1 – Un problème cumulatif avec un horizon fixé ($m = 9$) et 3 intervalles avec des capacités égales respectivement à 3, 2 et 3. Chaque activité requiert 2 unités de ressource. La première commence en $t = 0$ et finit en $t = 3$, la deuxième commence en $t = 2$ et finit en $t = 7$, la troisième commence en $t = 5$ et finit en $t = 7$.Il y a deux dépassements de capacité : un dans le premier intervalle au temps 2, et un dans le troisième intervalle aux temps 5 et 6.

Nous introduisons une nouvelle contrainte, SOFT-CUMULATIVE, qui étend le travail présenté dans [9] pour considérer des problèmes cumulatifs avec dépassements dans lesquels des capacités locales sont définies pour des intervalles de temps disjoints. Plusieurs mesures de violations peuvent être utilisées pour quantifier les dépassements de capacités locales et pour calculer une valeur d'objectif : on peut vouloir minimiser le dépassement le plus haut, ou minimiser la somme des aires en dépassement. Nous décrivons des problèmes concrets qui peuvent être modélisés en uti-

lisant SoftCumulative et d'autres contraintes additionnelles. Certaines de ces contraintes peuvent mener à des situations où les dépassements sont imposés dans certains intervalles de temps.

Notre contribution principale est un nouvel algorithme de filtrage associé à SoftCumulative, qui considère aussi les dépassements imposés. Il est décomposé en trois phases. La première phase est une adaptation de l'algorithme de Sweep en $O(n \cdot log(n))$, pour Cumulative [4], où n est le nombre d'activités. Dans notre cas, la complexité temporelle dépend aussi du nombre p de capacités locales définies par l'utilisateur : $O((n+p) \cdot log(n+p))$. Les avantages du sweep sont préservés : le profil est calculé et les variables de début des activités sont filtrées en un balayage, et la complexité ne dépend pas du nombre de points de temps. Pour réaliser un raisonnement énergétique [1] complémentaire au raisonnement du Sweep, basé sur le profil, dans une deuxième phase, nous adaptons l'algorithme d'Edge-Finding de P. Vilím's [13], en $O(k \cdot n \cdot log(n))$, où k est le nombre de demandes en capacité distinctes. Dans notre cas, la complexité temporelle est $O(p \cdot k \cdot n \cdot log(n))$. Dans les deux phases les bornes inférieures de l'objectif sont incluses dans les conditions de filtrage sans augmenter la complexité. La troisième phase est une propagation spécifique aux événements survenant sur les valeurs minimales de variables exprimant des violations de capacités locales, sans augmenter la complexité temporelle.

La Section 2 présente les pré-requis utiles pour comprendre nos contributions et définit SoftCumulative. La Section 3 présente des exemples motivant l'intérêt de notre travail. La Section 4 décrit l'algorithme de filtrage de SoftCumulative. La Section 5 présente une nouvelle stratégie de recherche dédiée, ainsi que les expérimentations effectuées en utilisant choco [1].

2 Définitions et Notations

Nous considérons un ensemble A d'activités non-préemptives, c'est à dire, des activités qui ne peuvent pas être interrompues. Une activité est définie par des variables. Etant donnée une variable x, $\underline{x}$ (resp. $\overline{x}$) représente la valeur minimum (resp. maximum) dans son domaine $D(x)$.

Définition 1 (Activité) *Une activité $a_i \in A$ est définie par quatre variables : sa_i représente son début ; da_i, sa durée ; ca_i, sa date de fin telle que $ca_i = sa_i + da_i$; et $ra_i \geq 0$, la quantité discrète de ressource consommée par a_i en tout point de temps entre son début et sa fin.*

1. Déductions basées sur la comparaison entre la ressource consommée et la ressource disponible.

Nous notons $\underline{e}(a_i)$ *l'énergie minimum* d'une activité $a_i \in A$, égale à $\underline{da_i} * \underline{ra_i}$. Etant donné $\Omega \subseteq A$, $\underline{e}(\Omega)$ est l'énergie minimum de Ω, égale à $\sum_{a_i \in \Omega} \underline{e}(a_i)$. Nous notons $\underline{mins}(\Omega) = \min_{a_i \in \Omega}(\underline{sa_i})$, $\underline{maxs}(\Omega) = \min_{a_i \in \Omega}(\overline{sa_i})$, $\overline{minc}(\Omega) = \max_{a_i \in \Omega}(\underline{ca_i})$, $\overline{maxc}(\Omega) = \max_{a_i \in \Omega}(\overline{ca_i})$. *L'horizon de temps m* est la fin maximum possible entre les activités. Nous imposons $\forall a_i \in A, ca_i \leq m$. Enfin, *une instantiation de A* est une affectation de toutes les variables définissant des activités dans A. En PPC, les problèmes cumulatifs peuvent être modélisés en utilisant la contrainte Cumulative [2]. Nous rappelons d'abord sa définition avant d'introduire SoftCumulative.

Définition 2 (Hauteur) *Etant donnée une ressource et une instantiation d'un ensemble A de n activités, à chaque point de temps t la* hauteur cumulée *h_t des activités passant par t est $h_t = \sum_{a_i \in A, sa_i \leq t < ca_i} ra_i$.*

Définition 3 (Cumulative) *Etant donnée une ressource avec une capacité limitée par capa et une instantiation d'un ensemble A de n activités, la contrainte Cumulative$(A, capa)$ est satisfaite ssi les deux contraintes suivantes sont satisfaites :*

- *C1 : Pour chaque activité $a_i \in A, sa_i + da_i = ca_i$, et*
- *C2 : A chaque point de temps t, $h_t \leq capa$.*

Nous définissons maintenant la nouvelle SoftCumulative, une contrainte représentant des problèmes cumulatifs avec un horizon fixé, des variables de coût exprimant des dépassements de capacité sur des intervalles de temps, et une variable objectif agrégeant les coûts.

Définition 4 (SoftCumulative) *Considérons une ressource avec une capacité limitée par capa et un ensemble A de n activités tel que $\overline{maxc}(A) \leq m$. Nous définissons :*

- *Une partition $P = [p_0, \ldots, p_{k-1}]$ de $[0, m[$ en p intervalles consécutifs tels que chaque p_j est défini par son début et sa fin : $p_j = [sp_j, ep_j[$.*
- *Une séquence de capacités locales $Loc = [lc_0, \ldots, lc_{k-1}]$ en correspondance une à une avec P, telle qu'une capacité locale $lc_j \leq capa$ est associée avec chaque intervalle p_j.*
- *Une séquence de variables de coût $Cost = [cost_0, \ldots, cost_{k-1}]$ en correspondance une à une avec P, telle qu'une variable $cost_j$ est associée avec l'intervalle p_j.*
- *Une variable objectif obj.*
- *Un indicateur $costC \in \{max, sum\}$ indiquant comment les variables Cost sont calculées (i.e. considérant le dépassement maximum ou la surface au-dessus).*

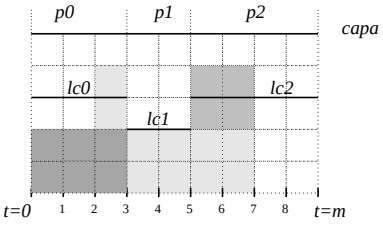


FIGURE 2 – Une instance de SOFTCUMULATIVE représentant le problème de la Figure 1. Si $costC = max$, $cost_0 = 1$, $cost_1 = 0$, $cost_2 = 1$ et si $objC = max$, $obj = 1$, si $objC = sum$, $obj = 2$. Si $costC = sum$, $cost_0 = 1$, $cost_1 = 0$, $cost_2 = 2$. Si $objC = max$, et si $objC = max$, $obj = 2$, si $objC = sum$, $obj = 3$

- *Un indicateur* $objC \in \{max, sum\}$ *indiquant comment la variable obj est calculée (i.e. considérant le maximum ou la somme des variables Cost).*

Etant donnée une instantiation de A, la contrainte SOFTCUMULATIVE$(A, capa, P, Loc, Cost, obj, costC, objC)$ *est satisfaite ssi les quatre conditions suivantes sont satisfaites :*

- *C1 et C2 (voir* Définition 3*).*
- *C3 :* $\forall j \in [0, k-1]$ *:* $cost_j = costC_{t \in p_j}(\max(0, h_t - lc_j))$
- *C4 : Une contrainte sur l'objectif :* $obj = objC_{j \in [0,k-1]}(cost_j)$.

Comparée à SOFTCUMULATIVESUM [9], notre contrainte partage l'horizon de temps en différents intervalles de longueur et capacité propres. Les dépassements sont quantifiés par une variable de coût sur chaque intervalle, selon $costC$, et une variable objectif agrégeant les coûts selon $objC$(voir la Figure 2).

Notation 1 *Pout un point de temps t donné,* $p_j(t) = [sp_j(t), ep_j(t)[$, $cost_j(t)$ *et* $lc_j(t)$ *sont, respectivement, l'intervalle* $p_j \in P$*, la variable* $cost_j \in Cost$ *et la capacité locale* $lc_j \in Loc$ *tels que* $t \in p_j$.

3 Problèmes pratiques

Les problèmes cumulatifs avec un horizon fixé peuvent impliquer des contraintes additionnelles sur les variables *Cost*, pour distinguer les solutions ayant un intérêt pratique des solutions qui seront rejetées par l'utilisateur final [9]. Nous présentons deux cas classiques.

Répartition équitable des dépassements. Dans de nombreuses applications, *e.g.*, des problèmes d'emplois du temps où les employés doivent effectuer des heures supplémentaires pour finir leurs activités dans des périodes surchargées, les dépassements de capacité doivent être équitablement répartis. Ce besoin a été mis en évidence dans [11]. Plus tard, différentes contraintes globales ont été introduites pour équilibrer des solutions [8, 10, 12]. Pour les cas simples, des contraintes de cardinalité classiques peuvent être utilisées. En utilisant SOFTCUMULATIVE, on peut définir une partition sur l'ensemble *Cost* et imposer que, dans chaque classe de la partition, le nombre minimum de variables de coût égales à 0 soit strictement positif (voir Figure 3).

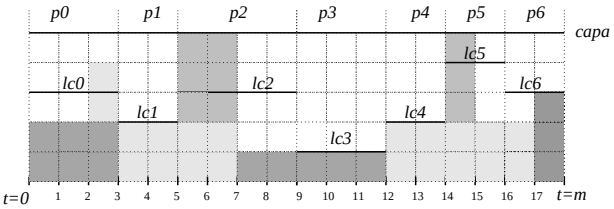


FIGURE 3 – Une solution satisfaisant des contraintes de cardinalité assitionnelles sur la partition $\{(cost_0, cost_1), (cost_2, cost_3), (cost_4, cost_5, cost_6)\}$ avec $costC = max$: au moins un coût égal à 0.

Lissage des variations de coût. Une requête fréquente consiste à limiter le nombre de variations importantes par rapport aux dépassements. Cela peut être fait en ajoutant, sur les variables *Cost*, une instance de SMOOTH$(N, tol, Cost)$ [3], où N est une variable et tol un entier. Elle impose que le nombre de fois où $|cost_{i+1} - cost_i| > tol$ soit égal à la valeur de N (i.e. N représente la limite). Par exemple, quand une entreprise embauche des intérimaires, leur nombre ne doit pas varier de manière trop importante d'un jour à l'autre. La Figure 3 est une solution satisfaisant à la fois SOFTCUMULATIVE et SMOOTH$(N, tol, Cost)$ avec $N = 1$ et $tol = 1$. Dans certaines applications, imposer des contraintes primitives de type $|cost_{i+1} - cost_i| \leq tol$ suffit.

Les dépassements mènent à des violations de capacités et les problèmes avec horizon fixés peuvent être vus comme des problèmes sur-contraints. Dans [9], les expérimentations montrent que, dans des problèmes sur-contraints avec contraintes additionnelles sur les variables représentant les violations, propager efficacement les contraintes additionnelles est indispensable pour résoudre les instances. Dans le cas de solutions équitablement réparties, une telle propagation correspond à des événements générés lorsque la valeur maximale d'une variables de coût est réduite. Plus généralement, lorsque l'on résout des problèmes sur-contraints, il est admis que les violations devraient être minimisées et ne sont jamais imposées [7]. L'exemple du lissage des variations de coût contredit cette suppo-

sition. Considérons le cas où la contrainte primitive $|cost_{i+1} - cost_i| \leq 1$ est imposée. Si $cost_i = 2$ alors il est nécessaire d'avoir $cost_{i+1} \geq 1$. En conséquence, pour être efficace dans tous les contextes possibles, le filtrage de SoftCumulative doit gérer les événements sur les valeurs minimales des variables *Cost*. La Section 4.5 présente un algorithme de filtrage dédié à de tels événements. Dans l'état de l'art [5], ce type de propagation est étudié pour SoftAllDifferent et SoftAllEqual.

4 Algorithmes de filtrage

4.1 Pré-requis

4.1.1 Sweep pour Cumulative

Le but est de réduire les domaines des variables de début en fonction du profil cumulatif, qui est construit à partir des parties obligatoires.

Définition 5 (Partie obligatoire [6]) *La* partie obligatoire $cp(a_i)$ *d'une activité* $a_i \in A$ *est l'intersection de tous les horaires possibles de* a_i*. Elle est définie par* $[\overline{sa_i}, \underline{ca_i}[$ *(de ce fait, elle est non vide ssi* $\overline{sa_i} < \underline{ca_i}$*), et une hauteur égale à* $\underline{ra_i}$ *on* $[\overline{sa_i}, \underline{ca_i}[$*, et nulle ailleurs.*

Définition 6 (Profil cumulatif) *Le* profil cumulatif *CumP est la consommation minimale cumulée, dans le temps, de toutes les activités. Pour un point de temps donné t, la hauteur de CumP en t égale à* $\sum_{a_i \in A, t \in [\overline{sa_i}, \underline{ca_i}[} \underline{ra_i}$*. C'est à dire, la somme des contributions de toutes les parties obligatoires passant par t.*

L'algorithme de sweep [4] déplace une ligne verticale Δ sur l'axe du temps, d'un événement à un autre. En un balayage, il construit le profil cumulatif et filtre les activités de manière à ne pas dépasser *capa*. Un *événement* correspond soit au début ou à la fin d'une partie obligatoire, soit à la date de début au plus tôt $\underline{sa_i}$ d'une activité $a_i \in A$. Tous les événements sont initialement calculés et triés par ordre croissant de leur date. La position de Δ est δ. A chaque pas de l'algorithme, une liste *ActToPrune* contient les activités à filtrer.

- Les événements de partie obligatoire sont utilisés pour construire *CumP* : Tous les événements de ce type à la date δ mettent à jour la hauteur sum_h du rectangle courant dans *CumP*, en ajoutant la hauteur de l'activité si c'est le début d'une partie obligatoire ou la soustrayant si c'en est la fin. Le premier événement de partie obligatoire avec une date strictement supérieure à δ donne la fin δ' du rectangle, finalement noté $\langle[\delta, \delta'[, sum_h\rangle$.
- Les événements correspondant aux dates de début au plus tôt $\underline{sa_i}$ tels que $\delta \leq \underline{sa_i} < \delta'$ ajoutent de nouvelles activités candidates au filtrage, en fonction de $\langle[\delta, \delta'[, sum_h\rangle$ et *capa* (activités passant par $\langle[\delta, \delta'[, sum_h\rangle$). Elles sont ajoutées à la liste *ActToPrune*.

Pour chaque $a_i \in ActToPrune$ n'ayant pas de partie obligatoire dans le rectangle $\langle[\delta, \delta'[, sum_h\rangle$, si sa hauteur est plus grande que $capa - sum_h$ alors on filtre sa date de début de manière à ce que a_i ne touche pas le rectangle courant de *CumP*. Si $\overline{ca_i} \leq \delta'$ alors a_i est retirée de *ActToPrune*. Après avoir filtré les activités, δ est mise à jour par δ'.

Règle de filtrage 1 *Si* $a_i \in ActToPrune$ *n'a pas de partie obligatoire dans* $\langle[\delta, \delta'[, sum_h\rangle$ *et* $sum_h + \underline{ra_i} > capa$ *alors* $]\delta - \underline{da_i}, \delta'[$ *peut être retiré de* $D(sa_i)$*.*

La complexité temporelle de Sweep est $O(n \cdot log(n))$ où n est le nombre d'activités.

4.1.2 Edge-Finding pour Cumulative

Cette section résume l'algorithme d'edge-finding en deux phases de Vilìm [13]. Il détecte d'abord les précédences induites entre les activités, et filtre ensuite les dates de débuts d'activités. Dans les deux phases, il utilise un raisonnement énergétique : Il compare la ressource requise par un ensemble d'activités dans un intervalle donné $I = [a, b[$ de points de temps avec la ressource disponible dans cet intervalle. Cette ressource disponible est donnée par la capacité *capa* et la taille de l'intervalle.

Définition 7 (Aire disponible). *Etant donné un intervalle de temps* $I = [a, b[$*, nous notons* $Area(a, b)$ *la ressource maximale disponible, égale à* $(b-a) * capa$*.*
(Enveloppe énergétique). *Soit* Ω *un ensemble d'activités tel que* $\Omega \subseteq A$*.* L'enveloppe énergétique *de* Ω *est :* $Env(\Omega) = \max_{\Theta \subseteq \Omega}(Area(0, \underline{mins}(\Theta)) + \underline{e}(\Theta))$.
(Précédence). *Une activité* $a_i \in A$ finit avant la fin *d'une activité* $a_j \in A$ *ssi, dans toutes les solutions,* $ca_i \leq ca_j$*. Cette relation est notée* $a_i \ll a_j$*. Elle peut être étendue à un ensemble d'activités* Ω *:* $\Omega \ll a_j$*.*

Pour détecter les précédences par rapport à une activité a_j, nous considérons toutes les activités a_i ayant une date de fin au plus tard $\overline{ca_i}$ plus petite ou égale à $\overline{ca_j}$.

Définition 8 (Coupe à gauche) *La* coupe à gauche *de A par une activité* $a_j \in A$ *est un ensemble d'activités telles que :* $LCut(A, a_j) = \{a_i \in A, \overline{ca_i} \leq \overline{ca_j}\}$.

Etant donné $\Omega \subseteq A$ (dans l'algorithme, systématiquement une coupe à gauche de A par une activité), l'enveloppe énergétique est utilisée pour calculer une

borne inférieure $lb(\overline{minc}(\Omega))$ pour la date de fin au plus tôt $\overline{minc}(\Omega)$ de Ω. D'après [13], $lb(\overline{minc}(\Omega)) = \lceil Env(\Omega)/capa \rceil$. Une fois les enveloppes énergétiques disponibles, il est possible de détecter les précédences.

Règle de précédence 1 *Si* $lb(\overline{minc}(LCut(A, a_j) \cup \{a_i\})) > \overline{ca_j}$, *alors* $LCut(A, a_j) \ll a_i$, *ce qui équivalent à : Si* $Env(LCut(A, a_j) \cup \{a_i\}) > Area(0, \overline{ca_j})$, *alors* $LCut(A, a_j) \ll a_i$.

A partir des précédences, l'algorithme de Vilìm met à jour les dates de début au plus tôt $\underline{sa_i}$ pour chaque activité qui finit nécessairement après la fin d'un ensemble d'activités Ω. Il est basé sur la notion de *compétition* : Ω *est en compétition* avec a_i ssi, $\underline{e}(\Omega) > Area(\underline{mins}(\Omega), \overline{maxc}(\Omega)) - \underline{ra_i} * (\overline{maxc}(\Omega) - \underline{mins}(\Omega))$.

Règle de filtrage 2 *Si* $\Omega \ll a_i$ *et* Ω *est en compétition avec* a_i *alors* $[\underline{sa_i}, \underline{mins}(\Omega) + \left\lceil \frac{\underline{e}(\Omega) - Area(\underline{mins}(\Omega), \overline{maxc}(\Omega)) + \underline{ra_i} * (\overline{maxc}(\Omega) - \underline{mins}(\Omega))}{\underline{ra_i}} \right\rceil [$ *peut être retiré de* $D(sa_i)$.

L'algorithme de Vilím utilise une structure de données arborescente pour calculer l'enveloppe énergétique $Env(\Omega)$ d'un ensemble donné $\Omega \subseteq A$, le Θ-tree, donnant une complexité temporelle globale de $O(k \cdot n \cdot log(n))$ où k est le nombre de $\underline{ra_i}$ distincts pour les activités a_i dans A.

4.2 Filtrage à partir des coûts maximaux

Des déductions peuvent être effectuées à partir d'événements sur les bornes supérieures de variables *Cost*. Les interactions avec la variable *obj* correspondant à des contraintes classiques de somme ou de max, nous nous focalisons sur le filtrage des variables de début.

4.2.1 Sweep pour SoftCumulative

Le filtrage ne dépend pas seulement de la capacité *capa*, mais également de la valeur maximale des variables *Cost*, et de *costC*. La notion de profil cumulatif est la même. Nous ajoutons une nouvelle classe d'événements : le début et la fin de chaque intervalle utilisateur $p_j \in P$. En conséquence, le rectangle défini par $\langle [\delta, \delta'[, sum_h \rangle$ a une capacité locale unique lc_j et correspond à une unique variable de coût $cost_j$. Nous avons deux propriétés : (1) Ces nouvelles activités ne modifient pas le calcul incrémental de sum_h dans le sweep. (2) Contrairement à la Règle de filtrage 1, la capacité maximale $capa_j$ à considérer pour un rectangle donné dépend de lc_j, $\overline{cost_j}$ et $costC$. Nous considérons d'abord $costC = max$.

Règle de filtrage 3 ($costC = max$) *Soit* $a_i \in ActToPrune$, *n'ayant pas de partie obligatoire enregistrée dans le rectangle* $\langle [\delta, \delta'[, sum_h \rangle$. *Si* $sum_h + \underline{ra_i} > lc_j + \overline{cost_j}$ *alors* $]\delta - \underline{da_i}, \delta'[$ *peut être retiré de* $D(sa_i)$.

Preuve 1 *Tout* a_i *dans* $ActToPrune$ *est tel qu'il existe au moins un point de temps* t *dans* $[\underline{sa_i}, \overline{ca_i}[$ *tel que* $\delta \leq t < \delta'$. *Par les Définitions 2 et 6, la hauteur* h_t *de toute solution étendant l'instantiation partielle courante est telle que* $h_t \geq sum_h + \underline{ra_i}$. *Alors, si* $sum_h + \underline{ra_i} - lc_j > \overline{cost_j}$, *la contrainte C3 de la Définition 4 est violée. Le même raisonnement s'applique en chaque point de temps de* $[\delta, \delta'[\cap [\underline{sa_i}, \overline{ca_i}[$. □

Si $costC = sum$, dans chaque intervalle utilisateur p_j, $cost_j$ correspond à l'aire dépassant la capacité locale lc_j dans cet intervalle. Rappelons que dans un intervalle donnée $[\delta, \delta'[$ défini dans l'algorithme de sweep, la capacité locale lc_j et la hauteur du profil sum_h restent constants par définition. La consommation du profil est $sum_h * (\delta' - \delta)$ et l'aire disponible totale ne menant pas à un dépassement est $lc_j * (\delta' - \delta)$. Nous essayons d'ajouter une activité a_i qui a une date de début au plus tôt $\underline{sa_i}$ dans l'intervalle $[\delta, \delta'[$. Si a_i a une durée minimum telle que a_i peut être contenue dans $[\delta, \delta'[$, alors son énergie minimale dans cet intervalle est $\underline{e}(a_i)$. Sinon, sa date de fin au plus tôt est à l'extérieur de $[\delta, \delta'[$, et son énergie dans cet intervalle est $(\delta' - \underline{sa_i}) * \underline{ra_i}$. Nous comparons l'énergie totale $(\delta' - \delta) * sum_h + \min((\delta' - \underline{sa_i}) * \underline{ra_i}, \underline{e}(a_i))$ à la ressource maximale disponible $lc_j * (\delta' - \delta) + \overline{cost_j}$. Nous filtrons $\underline{sa_i}$ si $\overline{cost_j}$ est dépassée.

Règle de filtrage 4 ($costC = sum$) *Soit* $a_i \in ActToPrune$, *n'ayant pas de partie obligatoire enregistrée dans le rectangle* $\langle [\delta, \delta'[, sum_h \rangle$ *et telle que* $\delta \leq \underline{sa_i} < \delta'$.
Si $(\delta' - \delta) * sum_h + \min((\delta' - \underline{sa_i}) * \underline{ra_i}, \underline{e}(a_i)) > lc_j * (\delta' - \delta) + \overline{cost_j}$ *alors* $[\underline{sa_i}, \min(\overline{\delta'}, \delta' - \lfloor (\overline{cost_j} + (lc_j - sum_h) * (\delta' - \delta))/(\underline{ra_i}) \rfloor)[$

peut être retiré de $D(sa_i)$.

Preuve 2 *Etant donnée* $a_i \in A$ *telle que* $\delta \leq \underline{sa_i} < \delta'$, *supposons que* a_i *commence en* $t = \underline{sa_i}$. *Si* $\underline{ca_i} < \delta'$, *sa contribution énergétique minimale est* $\underline{e}(a_i)$ *dans* $[\delta, \delta'[$, *sinon, c'est* $\underline{ra_i} * (\delta' - t)$. *Par les Définitions 2 et 6,* $(\delta' - \delta) * \overline{h_t} \geq (\delta' - \delta) * (sum_h + \underline{ra_i})$ *et* $(\delta' - \delta) * (sum_h + \underline{ra_i}) \geq (\delta' - \delta) * sum_h + \min((\delta' - t) * \underline{ra_i}, \underline{e}(a_i))$. *Alors, si* $(\delta' - \delta) * sum_h + \min((\delta' - t) * \underline{ra_i}, \underline{e}(a_i)) - lc_j * (\delta' - \delta) > \overline{cost_j}$ *(condition de filtrage), la contrainte C3 de la Définition 4 est violée.* $\overline{cost_j} + (lc_j - sum_h) * (\delta' - \delta)$ *est l'aire restant disponible et* $\lfloor \frac{\overline{cost_j} + (lc_j - sum_h) * (\delta' - \delta)}{\underline{ra_i}} \rfloor$ *est le nombre*

de points de temps pouvant être pris par a_i dans $[\delta, \delta'[$ sans violer la condition de filtrage. a_i ne peut pas commencer avant δ', ce qui mène à $\min(\delta', \delta' - \lfloor(\overline{cost_j} + (lc_j - sum_h) * (\delta' - \delta))/(\underline{ra_i})\rfloor)$. □

Le nombre d'événements dans le sweep dépend à la fois du nombre d'intervalles utilisateur et du nombre d'activités. La complexité temporelle est $O((n+p) \cdot log(n+p))$.

4.2.2 Edge-Finding pour SoftCumulative

Pour étendre l'algorithme de Vilìms au cas de la SoftCumulative, nous considérons les intervalles P et les capacités locales *Loc* au lieu d'une capacité unique *capa*. La Définition 7 (aire disponible) doit être adaptée. Dans un intervalle $I = [a, b[$, la ressource disponible dépend des capacités locales et des valeurs maximales des variables *Cost*. Nous devons étudier étudier le cas où a et b sont dans le même intervalle utilisateur et le cas où $p_j(a) \neq p_j(b)$.

Définition 9 (Aire disponible) *Etant donné un intervalle de temps $I = [a, b[$, nous notons $Area(a, b)$ la ressource maximale disponible. Nous distinguons deux cas :*

- *Si $costC = max$*
  - *Si $p_j(a) = p_j(b)$, $Area(a, b) = (b-a) * (lc_j(a) + \overline{cost_j}(a))$*
  - *Sinon, $Area(a, b) = (ep_j(a) - a) * (lc_j(a) + \overline{cost_j}(a)) + (b - sp_j(b)) * (lc_j(b) + \overline{cost_j}(b)) + \sum_{p_i \in]p_j(a), p_j(b)[} (ep_i - sp_i) * (lc_i + \overline{cost_i})$*
- *Si $costC = sum$*
  - *Si $p_j(a) = p_j(b)$, $Area(a, b) = (b-a) * lc_j(a) + \overline{cost_j}(a)$*
  - *Sinon, $Area(a, b) = (ep_j(a) - a) * lc_j(a) + \overline{cost_j}(a) + (b - sp_j(b)) * lc_j(b) + \overline{cost_j}(b) + \sum_{p_i \in]p_j(a), p_j(b)[} ((ep_i - sp_i) * lc_i + \overline{cost_i})$*

Définition 10 (Aire libre) *Etant donné un intervalle de temps $I = [a, b[$, nous notons $FreeArea(a, b)$ la ressource maximale disponible sans créer de nouvelle augmentation de dépassements. Le calcul est similaire à la Définition 9 excepté que les valeurs maximales des variables de coût ("$\overline{cost_j}$" et "$\overline{cost_i}$") sont remplacées par* 0.

Les principes pour calculer l'enveloppe énergétique et la coupe à gauche (voir les Définitions 7 et 8) restent les mêmes que dans le cas Cumulative, excepté que l'aire disponible est maintenant calculée grâce à la Définition 9. De plus, la Règle de précédence 1 reste également la même. Pour prouver ceci, nous introduisons la notation suivante (il n'y a pas besoin de calculer explicitement cette quantité dans l'algorithme).

Notation 2 (*inf*) *Etant donné Ω un sous-ensemble d'activités de A, et son enveloppe énergétique $Env(\Omega)$, $inf(\Omega)$ est le plus grand index j tel que $Area(sp_0, sp_j) \leq Env(\Omega)$.*

Comme dans le cas de Cumulative, étant donné un ensemble d'activités $\Omega \subseteq A$ (qui est, dans l'algorithme, systématiquement une coupe à gauche), nous adaptons le calcul d'une borne inférieure $lb(\overline{minc}(\Omega))$ pour la date de fin au plus tôt $\overline{minc}(\Omega)$ dans le cas où $costC = max$.[2] Rappelons que $sp_{inf(\Omega)}$ est le début de l'intervalle utilisateur d'index $inf(\Omega)$.

Proposition 1 $lb(\overline{minc}(\Omega)) = sp_{inf(\Omega)} + \left\lceil \frac{Env(\Omega) - Area(sp_0, sp_{inf(\Omega)})}{lc_{inf(\Omega)} + \overline{cost_{inf(\Omega)}}} \right\rceil$

Preuve 3 *$lb(\overline{minc}(\Omega))$ est le premier point de temps avant lequel $Env(\Omega)$ peut rentrer, en commençant en sp_0 (voir [13]). Par la Définition 2, $sp_{inf(\Omega)} \leq lb(\overline{minc}(\Omega)) < sp_{inf(\Omega)+1}$. $Env(\Omega) - Area(sp_0, sp_{inf(\Omega)})$ rentre dans $[sp_{inf(\Omega)}, sp_{inf(\Omega)+1}[$ en un nombre minimum de points de temps $\lceil (Env(\Omega) - Area(sp_0, sp_{inf(\Omega)})) / (lc_{inf(\Omega)} + \overline{cost_{inf(\Omega)}}) \rceil$.* □

A partir de la Proposition 1, l'équivalence de la Règle de précédence 1 reste vraie.

Propriété 1 *Les deux conditions suivantes sont équivalentes : (1) $lb(\overline{minc}(LCut(A, a_j) \cup \{a_i\})) > \overline{ca_j}$. (2) $Env(LCut(A, a_j) \cup \{a_i\}) > Area(sp_0, \overline{ca_j})$.*

Preuve 4 *A partir de la Proposition 1 et de la Définition 7, avec $\Omega = LCut(A, a_j) \cup \{a_i\}$.* □

Avec le nouveau calcul de l'aire disponible, la notion de compétition et la Règle de filtrage 2 restent les mêmes qu'avec Cumulative. A chaque feuille du Θ-tree de Vilìm, l'aire disponible prend $O(p)$ au lieu de $O(1)$ (voir la Définition 9). Le calcul des noeuds internes reste le même que dans l'algorithme de Vilìm. Alors, la mise à jour des dates de début d'activités se fait en $O(n \cdot p)$, si on ajoute la détection des précédences, on a une complexité globale $O(p \cdot k \cdot n \cdot log(n))$.

4.3 Bornes inférieures des coûts et de l'objectif

Cette section montre comment on peut mettre à jour le minimum des variables de coût et comment nous pouvons calculer des bornes inférieures de l'objectif, à partir des domaines des variables d'activités. Ces bornes inférieures peuvent être utilisées pour mettre à jour $\underline{obj}$.

2. Nous donnons seulement la preuve dans le cas où $costC = max$. Dans le cas où $costC = sum$ la démonstration est la même, mis à part le calcul de $lb(\overline{minc}(\Omega))$ qui est égal à $lb(\overline{minc}(\Omega)) = sp_{inf(\Omega)} + \lceil \frac{Env(\Omega) - Area(sp_0, sp_{inf(\Omega)}) - \overline{cost_{inf(\Omega)}}}{lc_{inf(\Omega)}} \rceil$.

4.3.1 Dans l'algorithme de sweep

Les variables *Cost* peuvent être directement mises à jour dans l'algorithme de sweep, au cours du calcul du profil.

Règle de filtrage 5 *Considérons le rectangle courant* $\langle[\delta, \delta'[, sum_h\rangle$ *dans le sweep.*

- *Si* $(costC = max)$ *alors si* $sum_h - lc_j(\delta) > \underline{cost_j}(\delta)$ *alors* $[\underline{cost_j}(\delta), sum_h - lc_j(\delta)[$ *peut être retiré de* $D(cost_j(\delta))$.
- *Si* $(costC = sum)$ *alors si* $(\delta' - \delta) * (sum_h - lc_j(\delta)) > \underline{cost_j}(\delta)$ *alors* $[\underline{cost_j}(\delta), (sum_h - lc_j(\delta)) * (\delta' - \delta)[$ *peut être retiré de* $D(cost_j(\delta))$.

Preuve 5 *A partir des Définitions 5 et 6.* □

Les bornes inférieures pour la variable objectif sont : si $objC = sum$ alors $LB = \sum_{j\in[0,k-1]} \underline{cost_j}$ et si $objC = max$ alors $LB = \max_{j\in[0,k-1]}(\underline{cost_j})$. Ces bornes inférieures peuvent être calculées incrémentalement sans augmentation de complexité temporelle.

4.3.2 Dans l'algorithme d'Edge-Finding

Les déductions sur les valeurs minimales des variables *Cost* en utilisant l'edge-finding sont faibles. De ce fait, nous nous concentrons uniquement sur le calcul d'une borne inférieure pour l'objectif.

Pour chaque activité $a_i \in A$, l'énergie minimale $\underline{e}(LCut(A, a_i))$ est calculée dans l'algorithme d'Edge-finding. Cette énergie devrait tenir dans l'aire définie par l'intervalle $[sp_0, \overline{ca_i}[$ par la Définition 8. Sinon, cela implique des dépassements et est susceptible de modifier la valeur minimale de la variable objectif. Suivant cette procédure, pour chaque activité, nous calculons une borne inférieure spécifique $LB(a_i)$ pour la variable objectif. Pour garder un calcul simple de chaque $LB(a_i)$, la proposition suivante considère qu'il n'y a pas de limite maximale sur le coût de chaque intervalle. Nous notons $\mathcal{P}$ un ensemble contenant tous les intervalles utilisateurs p_j tels que $sp_j \leq \overline{ca_i}$.

Proposition 2 *Soit* a_i *une activité de* A, *et* $LCut(A, a_i)$ *la coupe à gauche de* A *par* a_i. *Si* $\underline{e}(LCut(A, a_i)) > FreeArea(sp_0, \overline{ca_i})$ *alors*

- *Si* $(costC = max$ *et* $objC = sum)$ *alors* $LB(a_i) = \lceil \frac{\underline{e}(LCut(A,a_i)) - FreeArea(sp_0, \overline{ca_i})}{\max_{p_j\in\mathcal{P}}(min(\overline{ca_i}, sp_{j+1}) - sp_j))} \rceil$.
- *Si* $(costC = max$ *et* $objC = max)$ *alors* $LB(a_i) = \lceil \frac{\underline{e}(LCut(A,a_i)) - FreeArea(sp_0, \overline{ca_i})}{\overline{ca_i} - sp_0} \rceil$.
- *Si* $(costC = sum$ *et* $objC = sum)$ *alors* $LB(a_i) = \underline{e}(LCut(A, a_i)) - FreeArea(sp_0, \overline{ca_i})$.
- *Si* $(costC = sum$ *et* $objC = max)$ *alors* $LB(a_i) = \lceil \frac{\underline{e}(LCut(A,a_i)) - FreeArea(sp_0, \overline{ca_i})}{|\mathcal{P}|} \rceil$.

Preuve 6 *Par la Définition 8,* $\underline{e}(LCut(A, a_i))$ *est l'énergie minimale nécessairement placée avant* $\overline{ca_i}$. *L'énergie maximale rentrant dans* $[sp_0, \overline{ca_i}[$ *sans mener à un dépassement est* $FreeArea(sp_0, \overline{ca_i})$ *par la Définition 10. De ce fait, si* $\underline{e}(LCut(A, a_i)) > FreeArea(sp_0, \overline{ca_i})$ *alors cela induit une borne inférieure sur* $\underline{obj}$. *nous considérons* $costC = max$ *et* $objC = sum$. *Sans perte de généralité, nous considérons que toute l'aire de dépassement peut être placée dans l'intervalle de longueur maximale* I *commençant en* sp_j, *c'est à dire que nous considérons* $\max_{p_j\in\mathcal{P}}(min(\overline{ca_i}, sp_{j+1}) - sp_j)$ *: si l'on considère un ou plusieurs intervalles additionnels ayant, par définition, une longueur plus petite ou égale à la longueur de* I, *cela mènerait nécessairement à une somme de dépassements supérieure ou égale (rappelons que nous ne considérons pas de limite supérieure sur les coûts).* $LB(a_i)$ *est égal à* $\lceil \frac{\underline{e}(LCut(A,a_i)) - FreeArea(sp_0, \overline{ca_i})}{\max_{p_j\in\mathcal{P}}(min(\overline{ca_i}, sp_{j+1}) - sp_j))} \rceil$. *Si* $costC = max$ *et* $objC = max$ *alors nous étalons l'aire de dépassement sur le plus grand intervalle possible, c'est à dire* $[sp_0, \overline{ca_i}[$. *Si* $costC = sum$ *et* $objC = sum$, *la borne inférieure est la même quelle que soit la répartition des dépassements entre les intervalles utilisateur. Si* $costC = sum$ *et* $objC = max$, *en étalant le dépassement* $\underline{e}(LCut(A, a_i)) - FreeArea(sp_0, \overline{ca_i})$ *sur les intervalles* $\mathcal{P}$ *on obtient le plus petit coût maximum possible.* □

Notons que considérer les limites maximales sur les coûts ne peut qu'augmenter les bornes inférieures calculées. Le calcul de $LB(a_i)$ est intégré dans l'algorithme d'edge-finding et est effectué en $O(p)$. Alors, la complexité globale en intégrant cet algorithme dans l'edge-finding est toujours $O(k \cdot p \cdot n \cdot log(n))$.

4.4 Intégration des bornes inférieures de l'objectif

Dans cette section, nous considérons que les procédure de filtrage 4.2.1 ont été appliquées. Notre but est de filtrer les variables d'activités en utilisant les bornes inférieures de la variable objectif calculées dans la Section 4.3. Nous effectuons cette intégration en deux phases. La première est une seconde procédure de balayage, tandis que la seconde est un raisonnement énergétique. Sans perte de généralité, nous considérons que $\underline{obj} = LB$.

4.4.1 Sweep

Dans cette section, nous utilisons la borne inférieure LB de l'objectif calculée dans la Section 4.3.1. Nous considérons des activités n'ayant pas de partie obligatoire dans le rectangle courant $\langle[\delta, \delta'[, sum_h\rangle$, et la capacité locale correspondante lc_j.

Rappelons que LB et $\underline{cost_j}$ ont été calculées dans la première phase de balayage. Alors, si $costC = max$ alors $\underline{cost_j} \geq max(sum_h - lc_j, 0)$, et si $costC = sum$ alors $\underline{cost_j} \geq (\delta' - \delta) * max(sum_h - lc_j, 0)$.

Nous essayons d'ajouter une activité a_i dans l'intervalle $[\delta, \delta'[$, et ensuite nous calculons le coût obtenu en considérant que a_i est programmée dans $[\delta, \delta'[$. Si le coût calculé est strictement supérieur à $\underline{cost_j}$, alors nous pouvons considérer une LB augmentée sous condition que a_i soit placée dans $[\delta, \delta'[$.

Définition 11 *Soit $a_i \in ActToPrune$, n'ayant pas de partie obligatoire enregistrée dans le rectangle $\langle[\delta, \delta'[, sum_h\rangle$, et p_j l'unique intervalle contenant $[\delta, \delta'[$. Nous définissons $cost_j^{a_i}$ tel que :*

- *Si($costC = max$) alors $cost_j^{a_i} = max(sum_h + \underline{ra_i} - lc_j, 0)$.*
- *Si($costC = sum$) alors nous ne considérons que les activités a_i telles que $\delta \leq \underline{sa_i} < \delta'$ et $cost_j^{a_i} = max((\delta' - \delta) * (sum_h - lc_j) + \min((\delta' - \underline{sa_i}) * \underline{ra_i}, \underline{e}(a_i)), 0)$.*

Règle de filtrage 6 *Soit $a_i \in ActToPrune$, n'ayant pas de partie obligatoire enregistrée dans le rectangle $\langle[\delta, \delta'[, sum_h\rangle$, et p_j l'unique intervalle contenant $[\delta, \delta'[$.*

- *Si($costC = max$ et $objC = sum$) alors si $LB + max(cost_j^{a_i} - \underline{cost_j}, 0) > \overline{obj}$ alors $]\delta - \underline{da_i}, \delta'[$ peut être retiré de $D(sa_i)$.*
- *Si$costC = max$ et $objC = max$) alors si $cost_j^{a_i} > \overline{obj}$ alors $]\delta - \underline{da_i}, \delta'[$ peut être retiré de $D(sa_i)$.*
- *Si($costC = sum$ et $objC = sum$) alors si $LB + max(cost_j^{a_i} - \underline{cost_j}, 0) > \overline{obj}$ alors $[\underline{sa_i}, \min(\delta', \delta' - \left\lfloor \frac{(\overline{obj} - LB + \underline{cost_j}) + (lc_j - sum_h) * (\delta' - \delta)}{\underline{ra_i}} \right\rfloor)[$*

  peut être retiré de $D(sa_i)$.
- *Si($costC = sum$ et $objC = max$) alors si $cost_j^{a_i} > \overline{obj}$ alors $[\underline{sa_i}, \min(\delta', \delta' - \left\lfloor \frac{\overline{obj} + (lc_j - sum_h) * (\delta' - \delta)}{\underline{ra_i}} \right\rfloor)[$*

  peut être retiré de $D(sa_i)$.

Preuve 7 *$cost_j^{a_i}$ représente le coût minimal dans un intervalle $[\delta, \delta'[\subseteq p_j$ si nous ajoutons l'activité a_i à $[\delta, \delta'[$. Par la Définition 4, l'augmentation dans LB est $max(cost_j^{a_i} - \underline{cost_j}, 0)$ si $objC = sum$. Si cette augmentation est supérieure à la marge autorisée par $\overline{obj}$, alors nous pouvons filtrer sa_i. Si $costC = max$, tout point de temps pris par a_i dans $[\delta, \delta'[$ peut être responsable de l'augmentation, donc $]\delta - \underline{da_i}, \delta'[$ peut être retiré de $D(sa_i)$. Si $costC = sum$, le $cost_j$ maximal autorisé par $\overline{obj}$ est $\overline{obj} - LB + \underline{cost_j}$. Nous filtrons ensuite l'intervalle de la même manière que dans la Règle de filtrage 4. Le raisonnement est similaire pour $objC = max$ sauf que nous ne mesurons pas une augmentation, mais nous comparons directement le coût induit avec $\overline{obj}$.* □

4.4.2 Raisonnement énergétique

L'algorithme d'Edge-Finding calcule, pour chaque activité $a_i \in A$, l'énergie minimale $\underline{e}(LCut(A, a_i))$. Nous considérons ensuite la borne inférieure $LB(a_i)$ de la variable objectif calculée dans la Section 4.3.2.

Nous essayons d'ajouter une activité a_j telle que $\underline{sa_j} < \overline{ca_i}$ et $a_j \notin LCut(A, a_i)$ dans l'intervalle $[\underline{sp_0}, \overline{ca_i}[$. Nous pouvons ensuite calculer l'énergie totale obtenue en considérant que a_j est placée dans l'intervalle $[\underline{sp_0}, \overline{ca_i}[$. Si l'énergie calculée moins l'aire disponible dans cet intervalle est strictement plus grande que $LB(a_i)$, alors nous pouvons considérer une $LB(a_i)$ augmentée sous condition que a_i soit placée dans $[\underline{sp_0}, \overline{ca_i}[$. Alors si cette borne inférieure est strictement plus grande que $\overline{obj}$, nous pouvons filtrer sa_i.

Définition 12 *Soient a_i and a_j deux activités telles que $\underline{sp_0} \leq \underline{sa_j} < \overline{ca_i}$. Nous définissons $e(a_i, a_j)$ comme l'énergie de a_j dans cet intervalle sous condition que a_j commence en $\underline{sa_j}$: $e(a_i, a_j) = min(\underline{e}(a_j), (\overline{ca_i} - \underline{sa_j}) * \underline{ra_j})$.*

Nous nous concentrons ici sur $costC = max$ et $objC = sum$. Les autres cas sont similaires.

Règle de filtrage 7 ($costC = max$ et $objC = sum$) *Soient a_i et a_j deux activités dans A telles que $\underline{sa_j} < \overline{ca_i}$ et $a_j \notin LCut(A, a_i)$, et $LB(a_i)$ la borne inférieure de obj calculée dans la Section 4.3.2. Si $LB(a_i) + \lceil \frac{e(a_i, a_j)}{\max_{p_k \in \mathcal{P}}(min(\overline{ca_i}, sp_{k+1}) - sp_k))} \rceil > \overline{obj}$ alors $[\underline{sa_j}, min(\overline{ca_i}, \overline{ca_i} - \lfloor \frac{(\overline{obj} - LB(a_i)) * (\max_{p_j \in \mathcal{P}}(min(\overline{ca_i}, sp_{j+1}) - sp_j))}{\underline{ra_j}} \rfloor[$ peut être retiré de $D(sa_j)$.*

Preuve 8 *$LB(a_i)$ ignore $a_j \notin LCut(A, a_i)$. A partir de la Proposition 2, la règle est vérifiée.* □

4.5 Filtrage à partir des minimums de coûts

Dans la plupart des cas, en PPC, nous filtrons les activités à partir des bornes supérieures des variables de coût. La Section 3 montre qu'effectuer des déductions à partir des augmentations sur les minimums des domaines des variables *Cost* peut avoir un intérêt. Nous introduisons une technique basée sur la notion de partie enveloppante qui est duale à la notion de partie obligatoire.

Définition 13 (Partie enveloppante). *La* partie enveloppante $ep(a_i)$ *d'une activité* $a_i \in A$*, est l'union de tous les placements possibles pour* a_i*. Elle est définie par l'intervalle* $[\underline{sa_i}, \overline{ca_i}[$ *et une hauteur égale à* $\overline{ra_i}$ *sur* $[\underline{sa_i}, \overline{ca_i}[$*, et nulle ailleurs.*
(Profil cumulatif enveloppant). *Le* profil cumulatif enveloppant *EnvP est la consommation de ressource cumulée maximale, dans le temps, de toutes les activités. Pour un point de temps donné t, la hauteur de EnvP en t est égale à* $\sum_{a_i \in A, t \in [\underline{sa_i}, \overline{ca_i}[} \overline{ra_i}$ *(somme des contributions de toutes les parties enveloppantes passant par t).*

Pour filtrer les activités à partir des valeurs minimales des variables de coût, nous appliquons une procédure de balayage sur le profil cumulatif enveloppant et sur le profil des parties obligatoires. Pour ce faire, nous proposons d'ajouter une nouvelle classe d'événements : le début et la fin de la partie enveloppante de chaque activité. Ces nouveaux événements ne modifient pas le calcul incrémental de sum_h dans le sweep. De plus, nous considérons une nouvelle hauteur sum_e, qui est la hauteur du profil cumulatif enveloppant dans le rectangle courant. Intuitivement, le principe du filtrage est le suivant : Pendant la construction de $EnvP$, à chaque événement égal au début d'un intervalle utilisateur sp_j nous vérifions si il y a un unique rectangle $\langle[\delta, \delta'[, sum_e\rangle$ dans $EnvP$ entre sp_j et ep_j tel que $sum_e \geq lc_j + \underline{cost_j}$. Si c'est vrai, nous filtrons $a_i \in A$ avec la règle suivante.

1. Retirer de $EnvP$ la contribution de a_i.
2. Filtrer les bornes de $D(sa_i)$ pour s'assurer que si a_i commence en $\underline{sa_i}$ ou $\overline{sa_i}$ alors à tout point de temps t entre 0 et m, $EnvP$ peut atteindre $\underline{cost_j(t)}$.

Règle de filtrage 8 ($costC = max$) *Considérons un rectangle* $\langle[\delta, \delta'[, sum_e\rangle$ *et un intervalle* p_j *tel que* $\langle[\delta, \delta'[, sum_e\rangle$ *soit l'unique rectangle satisfaisant* $sum_e \geq lc_j + \underline{cost_j}$*. Soit* $a_i \in ActToPrune$*, si* $sum_e - \overline{ra_i} < lc_j + \underline{cost_j}$ *alors :* $[\underline{sa_i}, \delta - \overline{da_i}]$ *et* $[\delta', \overline{sa_i}]$ *peuvent être retiré de* $D(sa_i)$*.*

Preuve 9 *Si* $sum_e - \overline{ra_i} < lc_j + \underline{cost_j}$ *alors* a_i *devrait intersecter* $[\delta, \delta'[$*, sinon dans toute solution étendant l'instantiation partielle courante, aucun point de temps t dans* $[\delta, \delta'[$ *ne satisfera* $h_t \geq lc_j + cost_j$*. La contrainte C3 de la Définition 4 sera violée.* □

Nous considérons que, pour chaque événement correspondant au début d'un intervalle utilisateur, nous essayons d'applique la Règle de filtrage 8 par rapport à l'intervalle utilisateur précédent. Cela peut être effectué en $O(1)$. Nous ajoutons $O(n)$ événements au sweep et l'existence et la position de l'unique rectangle $\langle[\delta, \delta'[, sum_e\rangle$ satisfaisant $sum_e \geq lc_j + \underline{cost_j}$ peuvent être maintenus en $O(1)$. La complexité globale ne change pas.

5 Stratégies de recherche et expérimentation

Nous avons conçu SCSTRATEGY, une stratégie de recherche spécifique au problème représenté par notre contrainte. Elle est basée sur deux règles de priorité. (1) Choix de variable : sélectionner l'ensemble des activités maximisant la hauteur, et parmi elles sélectionner la variable de début d'une activité maximisant la durée. (2) Choix de valeur : Etant donnée la variable choisie, calculer l'ensemble des points de temps minimisant l'augmentation d'objectif, grâce à un balayage. Trier cet ensemble en fonction de l'énergie laissée libre une fois l'activité ajoutée en ce point, dans un ordre croissant

Nous avons implémenté SOFTCUMULATIVE dans CHOCO [1], avec $costC = max$ et $objC = sum$. Nous avons généré aléatoirement des instances avec des activités de durées 1 à 4 et de hauteurs 1 à 3, des capacités locales entre 3 et 6 ainsi que des longueurs pour les intervalles utilisateur. Les coûts maximaux ont été fixés à 6. Chaque activité peut être placée entre 0 et l'horizon. Les résultats dans les tableaux sont des moyennes sur 50 instances pour chaque classe de problème, en utilisant un processeur Intel Core 2, avec 4 Go de RAM.

Le premier test évalue le passage à l'échelle de notre contrainte, avec 16 ou 32 intervalles utilisateur. Avec notre implémentation, utiliser uniquement l'algorithme de sweep pour trouver une première solution est l'approche la plus efficace par rapport au temps. La première table (au dessus) donne les résultats obtenus en débranchant l'algorithme d'edge-finding, et montre que SOFTCUMULATIVE peut être utilisée pour trouver un solution pour des problèmes impliquant 1000 activités. SCSTRATEGY donne généralement, et plus rapidement, une solution avec une meilleure valeur d'objectif ($\pm 20\%$), comparé à une heuristique assignant statiquement les variables de début avec une valeur aléatoire (la meilleure heuristique par défaut dans CHOCO avec nos instances). Comme la complexité temporelle du sweep dépend du nombre d'intervalles, trouver une solution prend plus de temps lorsqu'il y a plus d'intervalles.

Dans un second benchmark nous essayons de trouver des solutions optimales. La deuxième table montre clairement que, dans ce cas, utiliser l'edge-finding améliore le processus de résolution. Les instances impliquent 4 ou 8 intervalles utilisateur et des contraintes

# activités / # intervalles utilisateur	longueur max des intervalles utilisateur	# résolus (avec le meilleur *obj*)		temps moyen (# backtracks)	
		Random	SCSTRATEGY	Random	SCSTRATEGY
500 / 16	40	48 (1)	49 (47)	2.2s (51)	2.1s (46)
500 / 32	20	50 (0)	50 (50)	3.5s (59)	2.8s (11)
1000 / 16	80	49 (1)	39 (38)	31.7s (221)	22.4s (150)
1000 / 32	40	49 (1)	46 (45)	67.3s (382)	49.5s (136)

# activités / # intervalles utilisateur	longueur max des intervalles utilisateur	# prouvés infaisables		# résolus avec $obj = 0$		# résolus avec $obj \neq 0$		temps moyen (#backtracks) avec $obj \neq 0$	
		No EF	EF	No EF	EF	No EF	EF	No EF	EF
50 / 4	20	0	10	11	11	0	27	> 2mn	1.0s (2183)
50 / 8	10	0	6	9	9	0	32	> 2mn	8.1s (7512)
100 / 4	41	0	12	8	8	0	27	> 2mn	4.9s (5534)
100 / 8	21	0	2	8	8	0	33	> 2mn	21.8s (11494)
125 / 4	51	0	6	8	8	0	34	> 2mn	9.6s (8540)
125 / 8	25	0	2	5	5	0	36	> 2mn	47.5s (19058)

TABLE 1 – Au-dessus : temps et nombre de backtracks moyens pour trouver une première solution. Comparaison d'une stratégie Random avec SCSTRATEGY. # résolus est le nombre d'instances résolues avec, entre parenthèses, le nombre d'instances où la variable objectif est la plus petite parmi les instances qui peuvent être résolues par les deux stratégies. En-dessous : temps et nombre de backtracks moyens pour trouver une solution optimale avec des contraintes additionnelles. Concernant le nombre d'instances prouvées infaisables et résolues, "EF" signifie que nous avons utilisé l'algorithme d'edge-finding, et "No EF" signifie que nous l'avons débranché.

additionnelles. Pour définir ces contraintes, nous avons partitionné les coûts par classe de 4 variables, sur lesquelles nous avons imposé : (1) Au moins une variable de coût prend la valeur 0 dans chaque classe de la partition, et (2) $\forall i$ tel que $0 \leq i < |Cost| - 1$, $|cost_{i+1} - cost_i| \leq 2$. Les instances dans la deuxième table ont été testées en utilisant l'heuristique par défaut dans CHOCO, SCSTRATEGY n'étant pas adaptée pour prouver l'optimum. Cette deuxième table montre qu'en utilisant l'Edge-Finding nous pouvons trouver la solution optimale de problèmes comportant 125 activités.

6 Conclusion

Nous avons présenté une nouvelle contrainte pour résoudre des problèmes cumulatifs avec dépassements. Nous avons adapté un algorithme de sweep et l'algorithme d'edge-finding de Vilìm à notre contexte. Nous avons proposé une procédure de filtrage spécifique aux dépassements imposés de l'extérieur. Nous avons conçu une heuristique de recherche dédiée. Une de nos perspectives est de considérer également la relaxation des relations de précédence entre activités.

Références

[1] Choco : An open source Java CP library. `http://choco.mines-nantes.fr`, 2011.

[2] A. Aggoun and N. Beldiceanu. Extending CHIP in order to solve complex scheduling and placement problems. *Mathl. Comput. Modelling*, 17(7) :57–73, 1993.

[3] N. Beldiceanu and M. Carlsson. Revisiting the cardinality operator and introducing the cardinality-path constraint family. *Proc. ICLP*, 2237 :59–73, 2001.

[4] N. Beldiceanu and M. Carlsson. A new multi-resource *cumulatives* constraint with negative heights. *Proc. CP*, pages 63–79, 2002.

[5] E. Hebrard, B. O'Sullivan, and I. Razgon. A soft constraint of equality : Complexity and approximability. *Proc. CP*, 5202 :358–371, 2008.

[6] A. Lahrichi. The notions of Hump, Compulsory Part and their use in Cumulative Problems. *C.R. Acad. sc.*, t. 294 :209–211, 1982.

[7] J. Larrosa and T. Schiex. Solving weighted csp by maintaining arc consistency. *Artificial Intelligence*, 159(1-2) :1–26, 2004.

[8] G. Pesant and J.-C. Régin. Spread : A balancing constraint based on statistics. *Proc. CP*, pages 460–474, 2005.

[9] T. Petit and E. Poder. Global propagation of side constraints for solving over-constrained problems. *Annals OR*, 184(1) :295–314, 2011.

[10] T. Petit and J-C. Régin. The ordered distribute constraint. In *ICTAI*, pages 431–438, 2010.

[11] T. Petit, J-C. Régin, and C. Bessière. Meta constraints on violations for over constrained problems. *Proc. IEEE-ICTAI*, pages 358–365, 2000.

[12] P. Schaus, Y. Deville, P. Dupont, and J-C. Régin. The deviation constraint. *Proc. CPAIOR*, 4510 :260–274, 2007.

[13] P. Vilím. Edge finding filtering algorithm for discrete cumulative resources in $O(kn\log(n))$. *Proc. CP*, pages 802–816, 2009.

Traces génériques et contraintes, GenTra4CP revisité

Pierre Deransart

INRIA Domaine de Voluceau, Rocquencourt, BP 105, 78153 Le Chesnay Cedex
Pierre.Deransart@inria.fr

Résumé

Le format de trace générique GenTra4CP a été défini en 2004 avec l'objectif de devenir un format de trace standard pour l'observation des solveurs de contraintes sur domaines finis. Il n'a pratiquement plus été utilisé depuis. Cet article reprend en la généralisant l'idée de trace générique et en donne un cadre formel basé sur des transformations simples de traces. Il analyse alors les limites du format initial proposé, apporte quelques précisions, et montre l'intérêt qu'un traceur générique peut avoir pour développer des applications portables, ou pour des efforts de normalisation, en particulier des contraintes.

Abstract

The generic trace format GenTra4CP has been defined in 2004 with the goal of becoming a standard trace format for the observation of constraint solvers over finite domains. It has not been used afterwards. This paper defines the concept of generic trace formally, based on simple trace transformations. It then analyzes and occasionally corrects shortcomings of the proposed initial format and shows the interest that a generic tracer may bring for developing portable applications or for standardization puposes, in particular in the field of constraints.

1 Introduction

Suite au projet RNTL OADymPPaC [1], un format de trace générique, appelé GenTra4CP (Generic Trace for CP), a été proposé en 2004 pour spécifier des traces de résolution de CSP(FD) dans le but de pouvoir réaliser des outils portables d'analyse de comportement de solveurs. Ce format se présente comme une sorte de norme comportant une syntaxe précise des événements de trace incluant une DTD XML, et une sémantique opérationnelle partielle, dite sémantique observationnelle, qui est en fait une sémantique opérationnelle commune à un ensemble potentiel de solveurs sur les domaines finis.

Des traceurs conformes au standard ainsi défini ont été implantés sur quatre solveurs. Plusieurs outils d'analyse de comportement de la résolution et de stratégies de recherche, ont été mis au point dans quatre environnements différents, à partir de la trace générique GenTra4CP. Ils ont pu être utilisés avec succès après un travail d'adaptation minimal pour chacun d'eux dans plusieurs environnements. Pour autant, à cette époque, aucune caractérisation formelle de la nature "générique" du format de trace n'a pu être donnée. Si la réalisation d'outils à partir d'un format de trace ainsi défini ne posait semble-t-il aucun problème et si l'on avait obtenu un gain considérable en portabilité, il était pratiquement impossible d'apprécier a priori l'effort d'adaptation nécessaire pour que les solveurs puissent utiliser les outils ainsi développés. De plus il n'était pas toujours possible de se rendre compte précisément de ce que certains outils "observaient" effectivement une fois adaptés. Le format GenTra4CP n'a de fait pu être utilisé que dans le projet qui l'a défini.

Cet article tente de combler ces lacunes et reprend en la formalisant l'idée de trace générique. Il analyse formellement la nature générique du format GenTra4CP et les limites du format proposé qui ont pu faire obstacle à son utilisation. Il montre également l'intérêt qu'un traceur générique peut avoir pour des efforts de normalisation, en particulier des contraintes, en proposant une approche de la sémantique reposant sur une sémantique opérationnelle partielle basée sur les traces.

Après une partie préliminaire où l'on adopte une approche de la sémantique opérationnelle d'un processus observé basée sur ses traces, la section 3 introduit quelques relations d'abstraction entre traces

dans le but d'appréhender formellement le concept de trace générique et d'apporter une méthode de preuve de conformité d'un processus à la trace générique. La section 4 expose la mise en œuvre pratique de cette approche et son intérêt en développement de logiciels. La section 5 applique cette approche au cas de GenTra4CP en vérifiant en particulier la conformité d'un solveur. Les descriptions formelles utilisées sont extraites de [1] pour GenTra4CP et [7] pour les solveurs. Cela permet de mieux appréhender la force et les limites de cette approche telle qu'elle a pu être utilisée à l'époque. On peut alors établir un lien entre les efforts de standardisation des contraintes et la méthode de spécification à partir de trace générique (section 6).

2 Préliminaires

Un objet *trace* est constitué d'un état initial s_0 suivi d'une suite finie ou infinie d'*événements de trace* ordonnés, et notée $< s_0, \overline{e} >$. Un ensemble de traces est noté $\mathcal{T}$. Un *préfixe* (fini, de taille t) d'une trace $T =< s_0, \overline{e_n} >$ (finie ou infinie, ici de taille n supérieure ou égale à t) est une trace partielle $U_t =< s_0, \overline{e_t} >$ correspondant aux t premiers événements de T, avec en plus au début l'état initial. Un préfixe réduit à l'état initial est de taille 0. L'ensemble des préfixes de $\mathcal{T}$ sera noté $Pref(\mathcal{T})$, avec $\mathcal{T} \subseteq Pref(\mathcal{T})$.

Toute trace est décomposable en segments ne comportant que des événements de trace, sauf le premier (dans l'ordre chronologique) qui débute par un état. On notera une décomposition avec un opérateur associatif de concaténation dénoté par un point et d'élément neutre ϵ (séquence vide). Ainsi une trace $T = T_1.T_2$ correspond à la trace T_1 (préfixe de T) à laquelle est concaténée la séquence composée des événements T_2. Une trace peut être le reflet de transitions d'états et à chaque événement peut correspondre un nouvel état. Dans ce cas un segment peut également être noté sT_ts_t où s est l'état dans lequel la séquence T_t débute, et s_t l'état atteint (état après le dernier événement de la séquence). Un segment (ou préfixe) de taille nulle est donc réduit soit à la suite vide, soit à un état.

Un *domaine de traces* sur $\mathcal{T}$, $\mathcal{DT}_{\mathcal{T}}$, est un ensemble dont chaque élément est l'ensemble de tous les préfixes d'une ou plusieurs traces de $\mathcal{T}$. Un élément est clos par préfixe. Un tel ensemble est clos pour l'union et l'intersection, et étant donnés deux éléments de $\mathcal{DT}_{\mathcal{T}}$ inclus l'un dans l'autre, le "plus petit" contient l'ensemble de tous les préfixes de quelques traces du "plus grand". Cet ensemble forme un treillis complet $\mathcal{DT}_{\mathcal{T}}(\subseteq, \bot, \top, \cup, \cap)$ dont $\bot$ est l'ensemble vide et $\top = Pref(\mathcal{T})$.

Les traces sont destinées à décrire l'évolution de systèmes décrits par l'évolution de leur état et/ou des suites d'actions. On distinguera deux types de traces :

- les *traces virtuelles* (ensemble $\mathcal{T}^v$) dont les événements sont de la forme $e = (r, s)$ où r est un *type d'action* nommant une transition d'un état à un autre et s, dit *état virtuel*, le nouvel état (état d'arrivée de la transition) décrit par un ensemble de *paramètres*.
- les *traces effectives* (ensemble $\mathcal{T}^w$) dont les événements sont de la forme $e = (a)$ où a est un *état effectif* décrit par un ensemble d'*attributs*. La trace effective est destinée à coder les changements d'états virtuels d'une manière plus synthétique.

On donne ici une description simplifiée de la sémantique des traces, appelée sémantique observationnelle dans [3].

Definition 1 (Sémantique Observationnelle (SO)) *Une Sémantique Observationnelle est définie par la donnée du sextuplet $< S, R, A, T, E_l, I_l, S_0 >$, où*

- *S :* domaine des états virtuels,
- *R : ensemble fini des* types d'actions, *ensemble d'identificateurs étiquetant les transitions.*
- *A :* domaine des états effectifs,
- *T :* fonction de transition d'états $T : R \times S \to S$, *notée $T(r, s) = s'$ ou $T(r, s, s')$ si c'est une relation,*
- *E_l :* fonction locale d'extraction de trace $E_l : S \times R \times S \to A$,
- *I_l :* fonction locale de reconstruction $I_l : S \times A \to R \times S$,
- *$S_0 \subseteq S$, ensemble des* états initiaux.

Les fonctions locales d'extraction E_l et de reconstruction I_l peuvent être étendues en fonctions E (resp. I) entre traces virtuelles et effectives, et doivent vérifier la relation dite de *fidélité*, $I = E^{-1}$. Les fonctions étendues peuvent être définies à partir des fonctions locales :

$E(s_0e_1...e_i...) = s_0E_l(s_0, r_1, s_1)...E_l(s_{i-1}, r_i, s_i)...$ avec $E_l(s_{i-1}, r_i, s_i) = a_i$, and

$I(s_0a_1...a_i...) = s_0I_l(s_0, a1)...I_l(s_i, a_{i+1})...$ avec $I_l(s_{i-1}, a_i) = (r_i, s_i)$.

Les fonctions locales et de transition seront représentées par des ensembles de règles (figure 1).

La sémantique observationnelle d'un processus observé peut être considérée comme une abstraction d'une sémantique opérationnelle raffinée [2]. Cette relation d'abstraction peut être exprimée comme une relation entre domaines de traces, et aussi bien entre traces effectives que traces virtuelles. Du fait de la propriété de fidélité, la fonction d'abstraction D_w portant sur les traces effectives vérifie avec D_v, la fonction

d'abstraction entre les traces virtuelles, les relations suivantes : $D_v = E_c \circ D_w \circ I_d$ et $D_w = I_c \circ D_v \circ E_d$ ($\circ$ denote la composition de fonctions). Les deux formules se déduisent l'une de l'autre du fait de la propriété de fidélité.

Dans la suite, on supposera la propriété de fidélité vérifiée, quelque soit le niveau de description. Dans ce cas les fonctions d'abstraction et de reconstruction peuvent se déduire l'une de l'autre. Dans la suite, on utilisera donc les seules traces virtuelles pour établir les propriétés recherchées.

$$\textsf{reduce}\ \frac{< \mathcal{D}(v),\ \ S_e,\ \ A >}{< \mathcal{D}(v) - \Delta_v^c,\ \ S_e \cup \bar{a},\ \ A' >} \left\{ \begin{array}{l} \text{supprime } \Delta_v^c), \\ a \text{ réveille } c \\ (c,a) \in A, \\ A' = A - \{(c,a)\} \\ v \in \mathbf{var}(c), \\ \text{génère } \bar{a} \end{array} \right\}$$

$$\textsf{reduce}\ \frac{\left\{ \begin{array}{l} < \mathcal{D}(v),\ \ S_e,\ \ A' \cup \{(c,a)\} > \rightarrow \\ < \mathcal{D}'(v),\ \ S'_e,\ \ A' > \end{array} \right\}}{[\textsf{reduce},\ \ c,\ \ v,\ \ (S'_e - S_e),\ \ (\mathcal{D}(v) - \mathcal{D}'(v)),\ \ a]}\ \{\}$$

$$\textsf{reduce}\ \frac{[\textsf{reduce},\ \ c,\ \ v,\ \ \bar{a},\ \ \Delta_v^c,\ \ a]}{\left\{ \begin{array}{l} < \mathcal{D}(v),\ \ S_e,\ \ A > \rightarrow \\ < \mathcal{D}(v) - \Delta_v^c,\ \ S_e \cup \bar{a},\ \ A - (c,a) > \end{array} \right\}}\ \{\}$$

FIGURE 1 – Description de la transition reduce dans la SO de GenTra4CP

3 Sous-trace et trace dérivée

On considère ici quelques transformations de traces, relativement simples à identifier et suffisantes pour les besoins de cette approche. Comme on peut se contenter de définir les transformations sur les traces virtuelles, on n'utilise qu'une partie de leur SO, à savoir $< S, R, T, S_0 >$.

Les sous-traces dites "paramétriques" sont obtenues à partir d'un sous-ensemble de paramètres.

Definition 2 (Sous-trace paramétrique) *Etant donnés un ensemble de traces $\mathcal{T}$ spécifié par $< S, R, T, S_0 >$, si S' est un sous-ensemble de S défini par un sous-ensemble des paramètres de S qui ne dépend*[1] *d'aucun paramètre de $S - S'$, R' est un sous-ensemble de R utilisant ou modifiant ces seuls paramètres tels qu'aucun autre type d'action de $R - R'$ ne les modifie, S'_0 est la restriction de S_0 à S', et T' la restriction de T à S' et R', alors l'ensemble de traces $\mathcal{T}'$ spécifié par $< S', R', T', S'_0 >$ est sous-trace (paramétrique) de $\mathcal{T}$, noté $Sub_P(\mathcal{T}, \mathcal{T}')$.*

1. Un paramètre p dépend d'un autre p' si p' intervient dans le calcul d'une valeur de p dans une transition.

Nota : il est possible d'avoir $S' \subseteq S$ et $R' = R$ ($S - S'$ contient des paramètres redondants, c'est à dire qu'ils ne dépendent que des autres paramètres de S et peuvent ainsi être supprimés).

Definition 3 (Champ de dérivation et trace dérivée)

Etant donnés deux ensembles de traces $\mathcal{T}_c$ et $\mathcal{T}_d$, où $\mathcal{T}_c$ et $\mathcal{T}_d$ sont dits respectivement concret *et* dérivé, *l'ensemble de traces $\mathcal{T}_d$ est un champ de dérivation par D de $\mathcal{T}_c$ s'il existe une application $D : Pref(\mathcal{T}_c) \rightarrow Pref(\mathcal{T}_d)$, dite* dérivation *telle que*

pour tout préfixe fini dérivé t_d de taille n et pour tout préfixe concret t_c dont il est l'image, il existe une suite croissante de préfixes concrets $[t_c^0, t_c^1, ..., t_c^i, ..., t_c^{n-1}, t_c]$ (non nécessairement contigus), appartenant tous au domaine de D, et tels que

- $D(t_c^0) \in S_{0,d}$,
- $\forall i > 0$ *si* $D(t_c^i) = t_d^i$ *avec* t_d^i *préfixe de* t_d *constitué des i premiers événements , alors* $D(t_c^{i+1}) = t_d^{i+1}$.

Si D est surjective, l'ensemble $\mathcal{T}_d$ est dit trace dérivée *par D de $\mathcal{T}_c$, que l'on peut noter par la relation binaire $Drv_D(\mathcal{T}_c, \mathcal{T}_d)$.*

Nota : si une trace dérivée t_d est de taille nulle et si elle est image de t_c, alors t_c ne peut être décomposée.

D est une fonction partielle. Il est possible de la rendre totale en considérant que tous les éléments de $S_{0,c}$ ont une image dans $S_{0,d}$ et que tout préfixe entre t_c^i et t_c^{i+1} a pour image $D(t_c^i)$ [2].

Cette définition est illustrée par le diagramme 2.

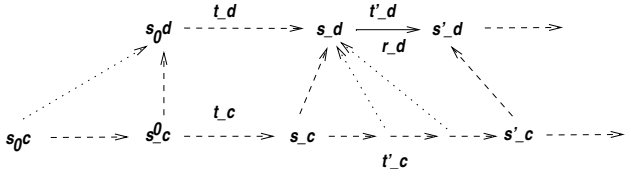


FIGURE 2 – Dérivation entre traces concrètes et dérivée (les flèches pointillées correspondent à la dérivation totale), t_x dénote un préfixe.

L'approche adoptée ici privilégie l'existence a priori d'un ensemble de traces, sans préjugé de la manière dont il a été produit ou spécifié. Elle favorise l'idée que la trace dérivée est le résultat de calculs sur les traces dites concrètes et formalisés par l'application D. A noter également que cette définition implique que la trace concrète qui a une image par D a une décomposition unique en préfixes du domaine de D.

2. Ainsi une trace, dont aucun préfixe de taille non nulle n'est dans le domaine de D, a pour image un état de $S_{0,d}$.

Si les traces dérivées peuvent être vues comme calculées à partir de traces concrètes, ces dernières peuvent être considérées comme des "implantations" ou des raffinements des premières dans un système où les états sont plus précis et où il y a des actions plus détaillées. Si une sous-trace peut être vue comme une dérivation, celle-ci s'en distingue par le fait qu'elle n'est pas nécessairement une sous-trace.

Propriétés 1

Etant données deux dérivations D_1 et D_2, si D_1 est surjective ou si D_2 est totale, $D_1 \circ D_2$ est une dérivation.

Une sous-trace au sens de la définition 2 est une trace dérivée.

Dans ce qui suit, on donne une méthode de preuve pour établir que deux ensembles de traces spécifiés par leur SO sont liés par une dérivation.

Definition 4 (Trace simulable)

Etant donnés deux ensembles de traces $\mathcal{T}_c$ (concret) et $\mathcal{T}_d$ (dérivé), définis respectivement avec $< S_c, R_c, T_c, S_{0,c} >$ et $< S_d, R_d, T_d, S_{0,d} >$, $\mathcal{T}_c$ est simulable *dans $\mathcal{T}_d$ si R_c et R_d sont en correspondance biunivoque par h, et s'il existe une application $d : S_c \to S_d$ qui vérifie :*

- $\forall s_0 \in S_{0,c}, d(s_0) \in S_{0,d}$.
- $\forall r_c \in R_c, s_c, s'_c \in S_c,\ T_c(s_c, r_c, s'_c) \Rightarrow \exists s_d, s'_d \in S_d, d(s_c) = s_d \wedge d(s'_c) = s'_d \wedge T_d(s_d, h(r_c), s'_d)$.

Théorème 1

Etant donnés deux ensembles de traces $\mathcal{T}_c$ (concret) et $\mathcal{T}_d$ (dérivé), tels que $\mathcal{T}_c$ est simulable dans $\mathcal{T}_d$, alors $\mathcal{T}_d$ est également un champ de dérivation pour $\mathcal{T}_c$ et la dérivation associée est totale.

Corollaire 1

Etant donnés deux ensembles de traces $\mathcal{T}$ et $\mathcal{T}'$ tels qu'il existe une sous-trace paramétrique de $\mathcal{T}$ simulable dans $\mathcal{T}'$, alors $\mathcal{T}'$ est un champ de dérivation pour $\mathcal{T}$.

4 Traces génériques

L'idée de trace générique répond à des besoins de spécification et de portabilité. On décrit un processus ou un algorithme par son comportement observable, c'est à dire par la trace d'opérations abstraites qu'il est censé implémenter. De plus on souhaite que cette description soit assez générale pour servir à la description d'une famille de processus, et enfin, on veut pouvoir l'utiliser dans certaines applications pour lesquelles cette information est suffisante. Ce peut être le cas d'applications comme le monitorage, le débogage, la construction d'outils de visualisation du comportement, ou même d'une application dont la trace a été définie a priori.

Definition 5 (Trace générique (TG)) *Etant donnée une famille de processus $p \in P$, à chacun desquels correspond un ensemble de traces $\mathcal{T}_p$, un ensemble de traces $\mathcal{T}_g$ est dit générique pour P, si, pour chaque processus p de la famille, il existe une dérivation de ses traces $\mathcal{T}_p$ par D_p, qui est une sous-trace de $\mathcal{T}_g$, soit*
$\forall p \in P,\ \exists\ \mathcal{T}$ *telle que* $Drv_{D_p}(\mathcal{T}_p, \mathcal{T}) \wedge Sub_P(\mathcal{T}_g, \mathcal{T})$.

Les principales questions qui se posent alors sont les suivantes.

- Comment s'assurer que l'on a correctement implanté la TG dans un processus particulier, ou, si on considère que la TG est une sorte de norme, en quoi une trace produite par ce processus est "conforme" à la TG ?
- La TG peut-elle être utilisée directement pour développer des applications dont on pourra garantir qu'elles fonctionneront avec tous les processus conformes ?
- La TG peut-elle être étendue à d'autres processus sans remettre en cause les implantations déjà réalisées et les applications existantes ?

Les questions précédentes ont alors les réponses possibles suivantes.

Conformité à la trace générique

On dira que la trace est générique pour un processus si elle satisfait la définition 5, c'est à dire qu'il existe une sous-trace de la trace générique qui est une dérivation de la trace (prédéfinie ou non) du processus. On dira dans ce cas que la trace du processus est "conforme" à la trace générique. Il est alors possible, soit d'implanter exactement la trace générique et donc que le traceur produise $Sub_P(T_p, T_g)$, soit d'implanter un traceur dans le processus p et prouver que $\exists \mathcal{T}', Drv_{D_p}(\mathcal{T}_p, \mathcal{T}') \wedge Sub_p(\mathcal{T}_g, \mathcal{T}')$.

Mise au point d'outils à partir de la trace générique

L'intérêt d'une trace générique est qu'elle facilite la mise au point d'outils susceptibles d'être utilisables sur tous les processus conformes. La mise au point s'effectue en considérant que l'outil utilise au moins des sous-traces de la TG correspondant à des processus décrits, voire toute la trace. Cela est possible à partir de la connaissance de la sémantique de la TG, c'est à dire de la connaissance de sa SO. L'utilisation pourra alors se faire à partir de chaque trace conforme des processus de la famille. L'effort d'adaptation de l'outil est alors réduit au minimum possible : celui de l'implantation de la dérivation D au niveau du processus (modification du traceur du processus pour produire la TG), ou au niveau de l'outil (adaptation de la trace du

processus pour reproduire la TG). La figure 3 illustre les deux possibilités.

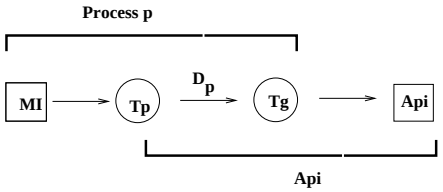


FIGURE 3 – Utilisations d'une trace générique : adaptation du processus ou de l'application

L'existence d'une spécification formelle de la trace générique permet également d'en réaliser un prototype qui s'ajoutera aux processus potentiellement décrits et devra répondre aux mêmes critères de conformité. Pour tester les outils en cours de mise au point on peut donc le faire, soit à partir d'un processus conforme, soit à partir d'un prototype conforme également. Tout ce que la démarche garantit est que si un outil est mis au point à partir d'un processus conforme dont une abstraction est la trace générique complète, il pourra fonctionner sur toutes les autres implantations conformes.

Extensions de la trace générique

Tant que les extensions de la TG préservent le fait que les processus conformes le sont par rapport à une sous-trace de la TG étendue, leur conformité par rapport à la TG étendue est préservée. Toute extension est telle que la trace originale reste une sous-trace générique. De cette manière les outils continuent de fonctionner sur les processus pour lesquels ils ont été réalisés.

5 La trace générique GenTra4CP

Dans le document final [1], la trace générique GenTra4CP est définie par une SO partielle [3] et par de nombreux attributs de la trace effective, très détaillé mais informellement décrits. Une syntaxe partielle sous forme d'une DTD XML décrivant chaque type d'événement de la trace effective est donnée.

On reprend ici le la description originale des états virtuels de GenTra4CP [1] (section 3.3.1) [4]

< < Début de Citation...

Definition 6 *(Solver State)*
A solver state is a 8-tuple : $\mathbb{S} = (\mathcal{V}, \mathcal{C}, \mathcal{D}, A, E, R, S_c, S_e)$
where : $\mathcal{V}$ *is the set of declared variables ;* $\mathcal{C}$ *is the set of declared constraints ;* $\mathcal{D}$ *is the function that assigns to each variable in* $\mathcal{V}$ *its domain (a set of values in the finite set D) ;* A *is the set of active couples of the form (constraint, solver event) ;* E *is the set of solved constraints ;* R *is the set of unsatisfiable (or rejected) constraints.* S_c *is the set of sleeping constraints ;* S_e *is the set of solver events to propagate ("sleeping events").*

A, S_c, E and R are used to describe four specific states of a constraint during the propagation stage : active, sleeping, solved or rejected.

The *store of constraints* constraints taken into account. The store is called σ in the following and defined as the partition $\sigma = \{c \mid \exists (c, a) \in A\} \cup S_c \cup E \cup R$. All the constraints in σ are defined, thus $\sigma \subseteq \mathcal{C}$. The set of variables involved in the c constraint is denoted by $\mathbf{var}(c)$. The predicate $false(c, \mathcal{D})$ (resp. $solved(c, \mathcal{D})$) holds when the constraint c is considered as unsatisfiable (resp. solved : it is universally true and does not influence further reductions any more) by the domains in $\mathcal{D}$.

The search is often described as the construction of a search-tree.

Definition 7 *(Search-Tree State)*
The search-tree is formalized by a set of ordered labeled nodes $\mathcal{N}$ *representing a tree, and a function* Σ *which assigns to each node a solver state. The nodes in* $\mathcal{N}$ *are ordered by the construction. Three kinds of nodes are defined and characterized by three predicates : failure leave (*$failed(\mathbb{S})$*), solution leave (*$solution(\mathbb{S})$*), and choice-point node (*$choice\text{-}point(\mathbb{S})$*). The last visited node is called* current node *and is denoted* n*. The usual notion of* depth *is associated to the search-tree : the depth is increased by one between a node and its children. The function* δ *assigns to a node* n *its depth* $\delta(n)$*.Therefore, the state of the search-tree is a quadruple :* $\mathbb{T} = (\mathcal{N}, \Sigma, \delta, n)$.

In the initial solver state, n_0 denotes the root of the search-tree and all the sets that are part of $\mathbb{S}$ are empty.

... Fin de Citation > >

Le reste de la description consiste à décrire chaque type d'événement de trace effective (appelé dans [7] "generic trace schema") en y introduisant d'autres attributs plus ou moins redondants (par exemple les identifiants interne et externes d'une contrainte).

On illustre l'approche formelle de la trace générique en analysant formellement une des "implantations" de GenTra4CP proposées dans [7]. Dans cet article, trois "spécialisations" de la trace génériques sont proposées pour 3 solveurs (GNU-Prolog, Choco et PaLM). Elles

3. SO partielle dans la mesure où seuls quelques paramètres sont considérés pour le modèle sémantique.

4. On utilise ici n au lieu de ν pour désigner le nœud courant de l'arbre de choix.

$$\text{new variable}\ \frac{\mathcal{V},\ \mathcal{D}}{\mathcal{V}\cup\{v\},\ \mathcal{D}\cup\{(v,\mathcal{D}_{v,i})\}}\ \left\{\begin{array}{l} v\notin\mathcal{V},\\ \mathcal{D}(v)=D_{v,i}\end{array}\right\}$$

$$\text{new constraint}\ \frac{\mathcal{C}}{\mathcal{C}\cup\{c\}}\ \left\{\begin{array}{l} c\notin\mathcal{C},\\ \mathbf{var}(c)\subseteq\mathcal{V}\end{array}\right\}$$

$$\text{post}\ \frac{A}{A\cup\{(c,\bot)\}]}\ \left\{\begin{array}{l} c\in\mathcal{C},\\ c\notin\sigma\end{array}\right\}$$

$$\text{choice point}\ \frac{\mathcal{N},\ \Sigma,\ \mathbb{S}}{\mathcal{N}\cup\{n\},\ \Sigma\cup\{(n,\mathbb{S})\},\ n}\ \left\{\begin{array}{l} ch\text{-}pt(\mathbb{S}),\\ n\notin\mathcal{N}\end{array}\right\}$$

$$\text{back to}\ \frac{\mathbb{S},\ \nu}{\Sigma(n),\ n}\ \left\{\begin{array}{l} n\neq\nu,\\ n\in\mathcal{N},\\ ch\text{-}pt(\mathbb{S})\end{array}\right\}$$

$$\text{solution}\ \frac{\mathcal{N},\ \Sigma,\ \mathbb{S}}{\mathcal{N}\cup\{n\},\ \Sigma\cup\{(n,\mathbb{S})\},\ n}\ \left\{\begin{array}{l} sol(\mathbb{S}),\\ n\notin\mathcal{N}\end{array}\right\}$$

$$\text{failure}\ \frac{\mathcal{N},\ \Sigma,\ \mathbb{S}}{\mathcal{N}\cup\{n\},\ \Sigma\cup\{(n,\mathbb{S})\},\ n}\ \left\{\begin{array}{l} flr(\mathbb{S}),\\ n\notin\mathcal{N}\end{array}\right\}$$

$$\text{remove}\ \frac{\sigma}{\sigma-\{c\}}\ \{\ c\in\sigma\ \}$$

$$\text{restore}\ \frac{\mathcal{D}(v)}{\mathcal{D}(v)\cup\Delta_v}\ \left\{\begin{array}{l} v\in\mathcal{V},\\ \Delta_v\cap\mathcal{D}(v)=\emptyset,\\ \Delta_v\subseteq\mathcal{D}_{v,i}\end{array}\right\}$$

FIGURE 4 – SO de GenTra4CP (contrôle, sans le paramètre δ)

Contrôle		
new variable	v,	$D_{v,i}$
new constraint	c	
post, remove	c	
restore	v,	Δ_v
choice point	n	
back to	n,	n'
solution, failure	n	

Propagation)		
reduce	c, v,	$\bar{a}$, Δ_v^c, a
suspend, solved	c	
reject	c,	a
awake	c,	a
schedule	c,	a

TABLE 1 – Attributs de la trace effective décrits dans la SO de GenTra4CP

consistent en une description de la sémantique opérationnelle de chaque solveur par une SO. On montre ici la conformité de la sémantique donnée pour PaLM [6]. Ce choix vient de ce que PaLM est, parmi les trois solveurs testés, le plus éloigné des principes qui sous-tendent le format GenTra4CP. La SO de GenTra4CP est rappelé dans les figures 4 et 5.

Afin de montrer que la SO de PaLM est bien conforme, on doit prendre en compte quelques propriétés de la trace générique, en particulier :

(G1) $sol(\mathbb{S}) \Rightarrow R=\emptyset$
(G2) $flr(\mathbb{S}) \Leftrightarrow R\neq\emptyset$
(G3) $(evtype = \text{reduce}) \Rightarrow R=\emptyset$
(G4) $(evtype = \text{awake}) \Rightarrow (R=\emptyset \wedge A=\emptyset)$
(G5) $(evtype = \text{schedule}) \Rightarrow (R=\emptyset \wedge A=\emptyset)$
On admet :

$$\text{reduce}\ \frac{<\mathcal{D}(v),\ S_e,\ A>}{<\mathcal{D}'(v),\ S'_e,\ A'>}\ \left\{\begin{array}{l} \mathcal{D}'(v)=\mathcal{D}'(v)-\Delta_v^c\\ \text{supprime } \Delta_v^c\\ (c,a)\in A\\ v\in\mathbf{var}(c)\\ \text{Red. gén. } \bar{a}\\ A'=A-(c,a)\\ S'_e=S_e\cup\bar{a}\end{array}\right\}$$

$$\text{suspend}\ \frac{<A,\ S_c>}{<A-\{(c,a)\},\ S_c\cup\{c\}>}\ \{(c,a)\in A\}$$

$$\text{solved}\ \frac{<A,\ E>}{<A-\{(c,a)\},\ E\cup\{c\}>}\ \left\{\begin{array}{l}(c,a)\in A,\\ solved(c,\mathcal{D})\end{array}\right\}$$

$$\text{reject}\ \frac{<A,\ R>}{<A-\{(c,a)\},\ R\cup\{c\}>}\ \left\{\begin{array}{l}(c,a)\in A,\\ false(c,\mathcal{D})\end{array}\right\}$$

$$\text{awake}\ \frac{<A,\ S_c>}{<A\cup\{(c,a)\},\ S_c-\{c\}>}\ \left\{\begin{array}{l} c\in S_c\\ a\in S_e\cup\{\bot\}\\ awcond(c,a)\end{array}\right\}$$

$$\text{schedule}\ \frac{<S_c,\ S_e>}{<S'_c,\ S'_e>}\ \left\{\begin{array}{l} c\in S_c,\\ e\in S_e,\\ action(c,a)\end{array}\right\}$$

FIGURE 5 – SO de GenTra4CP (propagation)

(P1) $dependence(c,a) \Leftrightarrow awcond(c,a)$
(P2) $select(a) \Rightarrow \exists c\in\mathcal{C}\ action(c,a)$
(P3) $\exists v\in\mathbf{var}(c), \mathcal{D}(v)=\emptyset \Rightarrow false(c,\mathcal{D})$

Théorème 2

La trace générique GenTra4CP, restreinte aux événements décrits aux figures 4 et 5 sauf back to *et* solved*, est une sous-trace de GenTra4CP, dérivée de la trace spécifiée pour PaLM (figures 6 et 7).*

La preuve est en annexe 1.

6 Trace générique et spécification de contraintes

Comme on l'a vu, une trace générique définie par uns SO est susceptible de constituer une forme de spécification d'une sémantique opérationnelle pour un ensemble de processus capables de produire une trace conforme. Cette approche peut être appliquée aux contraintes. La question est alors de pouvoir spécifier une trace générique qui couvrirait l'ensemble des contraintes que l'on souhaite décrire, c'est à dire qui couvrirait aussi bien différents types de contraintes (simples, globales, ...) , différents domaines (FD, intervalle, ...), différentes formes de solveurs (CSP, SAT ou règles, comme CHR), différents niveaux (algorithmes, modules, modélisation) que différents aspects (langages, interactions, interfaces, ...).

On se limite ici au cas CSP. Chaque contrainte a une sémantique déclarative définie par la relation qu'elle représente sur son domaine. La trace générique peut

$$\text{new variable} \quad \texttt{idem GenTra4CP}$$

$$\text{new constraint} \quad \texttt{idem GenTra4CP}$$

$$\text{post} \quad \texttt{idem GenTra4CP}$$

$$\text{choice point} \quad \texttt{idem GenTra4CP}$$

$$\text{solution} \ \frac{\mathcal{N},\ \Sigma,\ \mathbb{S}}{\mathcal{N}\cup\{n\},\ \Sigma\cup\{(n,\mathbb{S})\},\ n} \left\{ \begin{array}{l} sol(\mathbb{S}), \\ n\notin\mathcal{N} \end{array} \right\}$$

$$\text{failure} \ \frac{\mathcal{N},\ \Sigma}{\mathcal{N}\cup\{n\},\ \Sigma\cup\{(n,\mathbb{S})\},\ n} \left\{ \begin{array}{l} n\notin\mathcal{N}, \\ R\neq\emptyset \end{array} \right\}$$

$$\text{remove} \quad \texttt{idem GenTra4CP}$$

$$\text{restore} \ \frac{\mathcal{D}(v),\ Q_t,\ \mathcal{E}}{\mathcal{D}(v)\cup R_v,\ Q_t\cup\bar{a},\ \mathcal{E}-E} \left\{ \begin{array}{l} v\in\mathcal{V}, \\ R_v\subseteq\{d\in\mathbb{D}|\mathcal{E}(v,d)\cap\sigma\neq\emptyset\}, \\ E=\{\mathcal{E}(v,d)|d\in R_v\}, \\ \bar{a} \text{ actions de restauration de } \mathcal{D}(v) \end{array} \right\}$$

FIGURE 6 – SO de la trace de PaLM [7] (contrôle)

$$\text{reduce} \ \frac{\mathcal{D}(v),\ Q_t,\ \mathcal{E}}{\left\{ \begin{array}{l} \mathcal{D}(v)-\Delta_v^{c_a},\ Q_t\cup\{\bar{a}\}, \\ \mathcal{E}\cup\{(v,d,C)\,|\,d\in\Delta_v^{c_a}\} \end{array} \right\}}$$

$$\left\{ \begin{array}{l} v\in\mathbf{var}(c), \\ A=\{(c,a)\}, \\ \Delta_v^{c_a}\neq\emptyset \text{ set of inconsistent values for } v, \\ R=\emptyset, \\ C\subseteq\sigma \text{ explains the removal of } \Delta_v^{c_a} \text{ from } \mathcal{D}(v), \\ \text{The reduction generates } \bar{a} \end{array} \right\}$$

$$\text{suspend} \ \frac{<A,\ S_c>}{<\emptyset,\ S_c\cup\{c\}>} \ \{A=\{(c,a)\}\}$$

$$\text{reject} \ \frac{<A,\ R>}{<\emptyset,\ R\cup\{c\}>} \left\{ \begin{array}{l} A=\{(c,a)\}, \\ v\in\mathbf{var}(c), \\ \mathcal{D}(v)=\emptyset \end{array} \right\}$$

$$\text{awake} \ \frac{S_c,\ A}{S_c-\{c\},\ \{(c,a)\}} \left\{ \begin{array}{l} A=\emptyset, \\ c\in S_c, \\ R=\emptyset, \\ a\in Q_h\cup\{\bot\}, \\ dependence(c,a) \end{array} \right\}$$

$$\text{schedule} \ \frac{Q_h,\ Q_t}{\{a\},\ Q_t-\{a\}} \left\{ \begin{array}{l} select(a), \\ A=\emptyset, \\ a\in Q_t, \\ R=\emptyset \\ S_c\neq\emptyset \end{array} \right\}$$

FIGURE 7 – SO de la trace de PaLM [7] (propagation)

alors constituer une description des effets possibles de chaque contrainte prise isolément et dans un ensemble, indépendamment de l'algorithme particulier qui l'implante. Dans ce sens une telle sémantique joue le rôle d'une description à minima de ce que l'on doit pouvoir observer du comportement d'un ensemble de contraintes. Elle peut alors être utilisée pour définir a priori toutes sortes d'interfaces, en particulier pour la modélisation de problèmes.

En pratique, et c'est ce qui a été fait pour GenTra4CP, on va partir de la définition d'une trace effective dont la signification pourra être donnée par une fonction de reconstruction. On la complétera avec une SO éventuellement partielle, spécifiant un ensemble de traces acceptables à partir desquels le comportement d'un ensemble de contraintes pourra être simulé et et qui pourra servir à la mise au point d'applications.

Nous illustrons cette démarche de sémantique générique, avec un exemple simple, en montrant deux traces génériques du même problème obtenues avec GNU-Prolog et PaLM. Les deux solveurs ont été instrumentés pour produire la trace générique pour CSP(FD) et celle-ci peut être "comprise" conformément à la reconstruction décrite à la figure 8.

Les deux traces (figure 9) correspondent à la résolution de (syntaxe GNU-Prolog) `fd_element_var(I,[2,5,7],A), (A#=I ; A#=2)` qui admet une seule solution[5]. La sémantique déclarative de cette contrainte (toutes les variables, y compris celles de la liste, portent sur des domaines finis) peut se formuler : `fd_element_var(I, L, V)` (`L` liste) contraint `V` à être égale au `I`ème élément de `L`. En d'autres termes tous les triplets tels que $i \in$ `I`, $u \in$ `L`(i), $v \in$ `V` et $u = v$ sont valides. L'intervalle `[`*a*`-`*b*`]` dénote *de a à b* et `[`*a*`,`*b*`]`, *a et b*.

On peut observer[6] que les traces produites présentent des différences (entre autres) :

- le domaine de `I` qui n'est pas le même pour GNU (`[1-3]`) et pour PaLM (`[0-2]`) ;
- l'ordre et l'importance des retraits de valeurs et l'ordre du choix des variables dont on modifie le domaine ;
- les arbres de choix qui sont différents ;
- une variable spécifique qui apparaît dans la trace de PaLM (`v-1`).

Certaines variations ne sont pas significatives sur le

5. En fait PaLM produit des "raccourcis" tels qu'une séquence de reduce suspend schedule awake ne comporte que reduce awake. Un tel raccourci n'a pas de sémantique dans GenTra4CP (qui pourrait cependant être adaptée) et indique simplement que la sémantique de PaLM décrite dans [7] n'était pas tout à fait conforme à la trace générique.

6. GenTra4CP produit les traces dans un format XML, lisible mais trop verbeux. La présentation utilisée ici est une traduction plus concise.

$$\text{new variable}\ \frac{[\text{new variable},\quad v,\quad D_{v,i}]}{\left\{\begin{array}{l}<\mathcal{V},\ \mathcal{D}>\rightarrow\\ <\mathcal{V}\cup\{v\},\ \mathcal{D}\cup\{(v,D_{v,i})\}>\end{array}\right\}}\ \{\}$$

$$\text{new constraint}\ \frac{[\text{new constraint},\quad c]}{\left\{\begin{array}{l}<\mathcal{C}>\rightarrow\\ <\mathcal{C}\cup\{c\}>\end{array}\right\}}\ \{\}$$

$$\text{post}\ \frac{[\text{post},\quad c]}{\left\{\begin{array}{l}<A>\rightarrow\\ <A\cup\{(c,\bot)\}>\end{array}\right\}}\ \{\}$$

$$\text{choice point}\ \frac{[\text{choice point},\quad n]}{\left\{\begin{array}{l}<\mathcal{N},\ \Sigma,\ \mathbb{S}>\rightarrow\\ <\mathcal{N}\cup\{n\},\ \Sigma\cup\{(n,\mathbb{S})\},\ n>\end{array}\right\}}\ \{\}$$

$$\text{reduce}\ \frac{[\text{reduce},\quad c,\quad v,\quad \bar{a},\quad \Delta_v^c,\quad a]}{\left\{\begin{array}{l}<\mathcal{D}(v),\ S_e,\ A>\rightarrow\\ <\mathcal{D}(v)-\Delta_v^c,\ S_e\cup\bar{a},\ A-(c,a)>\end{array}\right\}}\ \{\}$$

$$\text{suspend}\ \frac{[\text{suspend},\quad c,\quad a]}{\left\{\begin{array}{l}<A,\ S_c>\rightarrow\\ <A-\{(c,a)\},\ S_c\cup\{c\}>\end{array}\right\}}\ \{\}$$

$$\text{awake}\ \frac{[\text{awake},\quad c,\quad a]}{\left\{\begin{array}{l}<A,\ S_c>\rightarrow\\ <A\cup\{(c,a)\},\ S_c-\{c\}>\end{array}\right\}}\ \{\}$$

FIGURE 8 – Quelques règles de la SO de GenTra4CP (reconstruction)

```
 1[0]choice point node(0)
 2[1]newVariable v1 [0-mx]
 3[1]newVariable v2 [0-mx]
 4[1]newConstraint c1
fd_element([v1,[2,5,7],v2])
 5[1]post c1
 6[1]reduce c1 v1 [0,4-mx]
 7[1]reduce c1 v2
   [0-1,3-4,6,8-mx]
 8[1]suspend c1
 9[1]choice point node(1)
10[2]newConstraint c4
    x_eq_y([v2,v1])
11[2]post c4
12[2]reduce c4 v2 [5,7]
13[2]reduce c4 v1 [1,3]
14[2]suspend c4
15[2]schedule v2 dom
16[2]awake c1
17[2]reject c1
18[2]failure node(2)
...
```

```
 0[0]newVariable v0 I [0-mx]
 1[0]newVariable v1 A [0-mx]
 2[0]newConstraint c0
     element(I,[2,5,7],A)
 3[0]post c0
 4[0]suspend c0
 5[0]awake c0
 6[0]reduce c0 v0 [3-mx] max
 7[0]reduce c0 v1 [0,1] min
 8[0]reduce c0 v1 [8-mx] max
 9[0]suspend c0
10[0]newConstraint c1 eq(I,A)
11[0]post c1
12[0]suspend c1
13[0]awake c0 (v0,max)
14[0]reduce c0 v1 [2-7] empty
15[0]reject c0 empty
16[0]failure
17[0]newVariable v-1 I [0-1]
18[0]reduce c2 v-1 [0,1] empty
...
```

FIGURE 9 – Traces effectives partielles de GNU-Prolog et PaLM pour l'exemple donné. Le deuxième attribut indique la profondeur de l'arbre de choix (paramètre δ ignoré ici)

plan sémantique (renommages, extra-variables, ...), et les traces restent compréhensibles et donc utilisables avec le même modèle générique. Certaines variations (domaine de `I`, attributs particuliers, ...) nécessiteraient probablement d'être fixées, ou au moins précisées pour arriver à une forme de standard.

7 Discussion

Les sémantiques des traces présentées ici (MI) se situent dans le cadre de la "sémantique observable" de Lucas [9] ou la sémantique des traces partielles de Cousot [2]. Les paramètres des états virtuels sont, comme l'exprime Lucas, des "objets syntaxiques utilisés pour représenter le déroulement de mécanismes opérationnels". Les traces sont des représentations abstraites qui permettent de ne tenir compte que des aspects que l'on souhaite ici communs à un ensemble de processus dotés de leur propre sémantique. Le choix d'articuler différentes formes de traces correspond au besoin de rapprocher différentes approches pragmatiques : spécification formelle de sémantiques plus ou moins abstraites et manipulations empiriques de traces dans des systèmes à base de traces, comme dans [10]. Nous avons établi ici une méthode particulière pour prouver la conformité de la trace d'un processus à une trace générique, mais cette approche permet également d'établir des liens formels avec la théorie des traces [4].

Nous avons montré ici que la définition de la trace GenTra4CP relève bien d'un tel cadre théorique et nous avons caractérisé par des opérations relativement simples (sous-traces paramétriques et similarité) les liens formels existant entre les processus observés et la trace générique. Cette analyse nous a révélé des insuffisances dans la définition même de GenTra4CP comme l'absence de test formel de conformité des traces des solveurs. Simonis & al [11] définissent une trace "générique" pour une famille d'outils bien définie. Ils notent que la trace générique GenTra4CP contient trop de détails pour une spécification trop complexe. Dans le type d'applications concernées, les besoins d'information sont en effet limités et il peut paraître plus facile de réaliser une instrumentation ad-hoc d'un solveur, plutôt que faire l'effort d'implanter a priori une trace générique avec tous ses détails. Mais cela est probablement faux si l'on souhaite réaliser un véritable interface générique entre plusieurs sortes de solveurs et de nombreuses applications potentielles.

Cette étude montre que c'est plutôt la trop grande laxité du modèle adopté qui pourrait être en cause, bien des paramètres essentiels à la description sémantique restant optionnels, ce qui lui fait perdre une partie de son utilité. Une approche plus exigeante mais plus utile serait sans doute d'avoir plus d'événements de trace (éventuellement plus spécifiques) portant sur des paramètres communs non optionnels.

Par ailleurs, comme il a été observé dans la section 4, il appartient aux développeurs de chaque processus ou de chaque outil, de produire une (sous-)trace

générique ou d'adapter des outils, développés sur la base de cette trace, au processus particulier. L'investissement à réaliser se mesure par l'écart formel qu'il y a entre la trace propre du processus et la trace générique (formellement la dérivation, cf. figure 3). Il peut paraître plus facile d'implanter une trace ad-hoc plutôt que d'adapter un traceur ou un outil à une trace prédéfinie. Langevine et Ducassé ont bien montré [8] qu'une approche générique pouvait avoir plus d'avantages que d'inconvénients, mais elle s'apparente à un effort de standardisation.

Un tel effort ne peut résulter que de l'action d'une communauté importante, et non d'un petit groupe comme dans le cas de GenTra4CP. Le projet de standard [5] se concentre principalement sur la définition d'un interface Java englobant en particulier les types principaux de variables, contraintes unaires, binaires et globales et certaines capacités à trouver des solutions. La question sémantique ne peut cependant être évitée. Si la sémantique déclarative de contraintes simples ne pose guère de problème de spécification, il n'en est pas de même de la sémantique opérationnelle, sa précision éventuelle dépendant des usages que l'on souhaite en faire. L'approche présentée ici, fondée sur une sémantique de trace générique, peut constituer une voie dans la mesure où elle offre un cadre pour spécifier des effets directs et collatéraux, liés par exemple aux interactions de contraintes, indépendamment d'implantations particulières.

8 Conclusion

GenTra4CP a constitué une démarche innovante en utilisant une sémantique de traces partielle pour répondre à un double objectif de spécification de résolution de CSP (domaines finis) et de portabilité d'outils d'analyses. Un tel effort s'apparente à une démarche de standardisation, mais difficile à utiliser, en raison de ses limites (limites du groupe qui l'a réalisée, insuffisances techniques et limites du domaine des contraintes traitées).

Nous avons introduit un cadre formel simple basé sur la théorie des traces et l'interprétation abstraite pour expliquer la méthode de construction de traces génériques et montrer l'intérêt potentiel de cette approche pour rendre compte de sémantiques partielles de contraintes.

Obtenir une sémantique sous-forme de trace générique pour un ensemble significatif de contraintes simples ou globales représente certainement un effort considérable. Il semble cependant qu'une telle approche permettrait non seulement d'assurer une certaine portabilité d'applications potentielles, mais aussi d'aller vers des descriptions de traitements entre différentes approches de systèmes de gestion de connaissances mêlant contraintes et règles de déduction.

Références

[1] A. Agoun, T. Baudel, P. Deransart, M. Ducassé, J.-D. Fekete, and N. Jussien. Tools for Dynamic Analysis of Constraints Resolution (OADymPPaC). Technical report, Inria Rocquencourt and École des Mines de Nantes and INSA de Rennes and Université d'Orléans and Cosytec and ILOG, May 2004. Projet RNTL. `http ://contraintes.inria.fr/OADymPPaC`.

[2] P. Cousot and R. Cousot. Systematic design of program transformation frameworks by abstract interpretation. In *Proc. of POPL 2002*, pages 178–190, 2002.

[3] P. Deransart. Conception de Traces et Applications (vers une méta-théorie des traces), June 2011. Working document `http ://hal.inria.fr/`.

[4] V. Diekert and G. Rozenberg. *The Book of Traces*. World Scientific Publishing, Singapoore, 1995.

[5] Jacob Feldman. JSR-331, Java Constraint Programming API. Tr, Java Community Process, Cork Constraint Computation Centre, 2011.

[6] Narendra Jussien and Vincent Barichard. The PaLM system : explanation-based constraint programming. In *Proceedings of TRICS : Techniques foR Implementing Constraint programming Systems, a post-conference workshop of CP 2000*, pages 118–133, Singapore, September 2000.

[7] Ludovic Langevine, Pierre Deransart, and Mireille Ducassé. A generic trace schema for the portability of cp(fd) debugging tools. In K.R. Apt, F. Fages, F. Rossi, P. Szeredi, and Jozsef Vancza, editors, *Recent Advances in Constraints*, number 3010 in LNAI. Springer Verlag, May 2004.

[8] Ludovic Langevine and Mireille Ducassé. Design and implementation of a tracer driver : Easy and efficient dynamic analyses of constraint logic programs. *Theory and Practice of Logic Programming, Cambridge University Press*, 8(5-6), Sep-Nov 2008.

[9] Salvador Lucas. Observable Semantics and Dynamic Analysis of Computational Processes. Technical Report LIX/RR/00/02, Laboratoire d'Informatique LIX, 2000.

[10] Lotfi Sofiane Settouti. *Système à base de trace modélisées : Modèles et Langages pour l'exploitation de traces d'interactions*. PhD thesis, Université Claude Bernard - Lyon I, January 2011.

[11] Helmut Simonis, Paul Davern, Jacob Feldman, Deepak Mehta, Luis Quesada, and Mats Carlsson. A Generic Visualization Platform for CP. In Karen Petrie, editor, *Proceedings of the 16th International Conference on Principles and Practice of Constraint Programming*, St Andrews, Scotland, September 2010.

ANNEXE : Preuve du théoreme 2

Preuve 1

On montre qu'une sous-trace de PaLM (issue de la SO de PaLM) est simulable dans une sous-trace "PaLM" de GenTra4CP.

On considère la trace générique GenTra4CP, restreinte à tous ses événements décrits aux figures 4 et 5 sauf back to *et* solved. *La suppression de l'événement* back to *n'affecte pas la construction de l'arbre de recherche, mais seulement son parcours et celle de* solved *correspond à la suppression du paramètre E de l'état du solveur. De plus, on "oublie" le paramètre concernant les explications, ce paramètre n'étant pas formalisé de manière générique dans la sémantique de GenTra4CP.*

Selon la définition 2, la restriction au sous-ensemble d'événements considérés est bien une sous-trace de GenTra4CP.

La sous-trace de PaLM considérée ici consiste à simplement ignorer les explications. Cela n'apporte aucune restriction sur les ensembles de types d'action (définition 2).

Dans le format GenTra4CP et la sous-trace considérée, la partie contrôle $\mathbb{T}_g$ utilise 4 paramètres : $\mathcal{N}, \Sigma, \delta, \nu$, et la partie solveur $\mathbb{S}_g$ 8 paramètres : $\mathcal{V}, \mathcal{C}, \mathcal{D}, A, E, R, S_c, S_e$. Au total 12 paramètres.

Dans PaLM, la partie contrôle $\mathbb{T}_p$ utilise 5 paramètres : $\mathcal{N}, \Sigma, \delta, \nu, Q_t$, et la partie solveur $\mathbb{S}_p$ 9 paramètres : $\mathcal{V}, \mathcal{C}, \mathcal{D}, A, R, S_c, Q_h, Q_t, \mathcal{E}$. Au total 13 paramètres (Q_t est commun). On note les différences suivantes :

- *E, le sous-ensemble du "store" contenant les contraintes universellement satisfaites disparaît dans PaLM qui ne réalise pas de test de satisfaction ("entailment").*
- *l'ensemble S_e des événements à traiter du solveur PaLM est une queue ($S_e = Q_h \cup Q_t$) dont la tête Q_h contient l'événement courant pris en compte.*
- *L'état du solveur de PaLM contient un paramètre additionnel $\mathcal{E}$, fonction d'explication qui sert à mémoriser les "explications". E est une fonction partielle : $\mathcal{E} : \mathcal{V} \times \mathbb{D} \longrightarrow \mathcal{P}(\sigma)$* [7] *qui affecte à chaque retrait de valeur (v, d) ($v \in \mathcal{V}, d \in \mathcal{D}(v)$) un ensemble de contraintes non relaxées qui "explique" le retrait. Cette fonction partielle est mise à jour au cours des événements* reduce *et* restore.
- *A, dans PaLM, a au plus un élément.*

On montre alors que la sous-trace ainsi restreinte de PaLM est simulable dans la sous-trace "PaLM" GenTra4CP.

On utilise le théorème 1. On définit l'application d entre les états modifiés $\mathbb{T}_p \times \mathbb{S}_p$ et $\mathbb{T}_g \times \mathbb{S}_g$, soit : (on omet δ qui se déduit directement de $\mathcal{N}$)

$\mathcal{N}, \Sigma, \nu, \mathcal{V}, \mathcal{C}, \mathcal{D}, A, R, S_c, Q_h, Q_t$ et
$\mathcal{N}, \Sigma, \nu, \mathcal{V}, \mathcal{C}, \mathcal{D}, A, R, S_c, S_e$

de la manière suivante : identité pour les 9 premiers paramètres de PaLM $\mathcal{N}, \Sigma, \nu, \mathcal{V}, \mathcal{C}, \mathcal{D}, A, R, S_c$, puis $Q_h \cup Q_t = S_e$.

Les types d'actions ont les mêmes noms et leur ensemble est restreint à ceux indiqués sur les figures 6 et 7.

Les états initiaux $\mathbb{T}_{0,p} \times \mathbb{S}_{0,p}$ et $\mathbb{T}_{0,g} \times \mathbb{S}_{0,g}$ à considérer sont :
$\{rc_p\}, (rc_p, \mathbb{S}_{0,p}), rc_p, \emptyset_p, \emptyset_p, \emptyset_p, \emptyset_p, \emptyset_p, \emptyset_p, \emptyset_p, \emptyset_p$ et
$\{rc_g\}, (rc_g, \mathbb{S}_{0,p}), rc_g, \emptyset_g, \emptyset_g, \emptyset_g, \emptyset_g, \emptyset_g, \emptyset_g, \emptyset_g, \emptyset_g$

new variable, new constraint, post, choice point *et* remove *sont en correspondance puisque les règles sont identiques et les paramètres concernés sont les mêmes.*

Pour solution *et* failure, *il en est de même à condition d'utiliser les propriétés (G1) et (G2).*

Le cas de restore *est plus complexe. Néanmoins, si on ignore les explications et si on prend pour Δ_v, R_v ($\Delta_v = R_v$) pour la même variable v, les conditions associées à l'événement de Gentra4CP se déduisent des propriétés des explications (restitution de valeurs retirées donc inexistante dans l'état courant du domaine de v). Par contre il faut alors expliciter dans la transition GenTra4CP la mise à jour de S_e.*

reduce. *A $\Delta_v^{c_a}$ correspond Δ_v^c (ensemble des valaurs inconsistantes) de la trace générique. Par (G3) les propriétés $R = \emptyset$ se correspondent. Enfin comme $d(Q_h \cup Q_t) = S_e$, $d(Q_h \cup Q_t \cup \bar{a}) = S_e \cup \bar{a}$.*

suspend. *Dans les états initiaux en correspondance $(c, a) \in A$, et $A' = A - \{(c, a)\}$ dans les états finaux.*

reject. *Utilise (P3) pour les états initiaux, et les états finaux se correspondent.*

awake. *Utilise (P1) et (G4).*

schedule. *Utilise (P2) et (G5). S_c et S_e sont invariants dans la trace générique.*

7. $\mathcal{P}(\sigma)$: ensemble des parties du store σ (instance des contraintes qui sont dans A, S_c et R pour PaLM).

Domaine Consistance et Valeurs Interdites

Yves Deville[1] **Pascal Van Hentenryck**[2] **Jean-Baptiste Mairy**[1]

[1] ICTEAM, UCLouvain, Belgique
[2] Brown University, USA
{Yves.Deville,Jean-Baptiste.Mairy}@uclouvain.be pvh@cs.brown.edu

Abstract

Cet article [1] présente un nouvel algorithme de domaine consistance qui ne maintient pas les supports dynamiquement lors de la propagation, mais plutôt maintient les valeurs interdites. Nous introduisons l'algorithme optimal NAC4 (AC4 négatif), basé sur cette idée. Nous montrons en outre que le maintien de valeurs interdites permet à l'algorithme générique AC5 d'assurer la domaine consistance en temps $O(ed)$ pour les catégories de contraintes dans lequel le nombre de supports est de $O(d^2)$, mais où le nombre de valeurs interdites est de $O(d)$. Le papier montre également comment les valeurs interdites et les supports peuvent être utilisés conjointement pour assurer la domaine consistance sur des combinaisons logiques de contraintes ainsi que pour calculer la validité et l'implication de contraintes. Les résultats expérimentaux montrent les avantages de l'exploitation communes des supports et des valeurs interdites.

1 Introduction

En programmation par contraintes, la propagation vise à réduire l'espace de recherche sans retirer de solutions. L'algorithme de propagation considère chaque contrainte individuellement et se termine lorsque aucune contrainte ne peut être utilisée pour réduire les domaines des variables. La propagation idéale pour une contrainte est la *consistance de domaine*, aussi connu sous la consistance d'arc. Elle supprime du domaine de chaque variable toutes les valeurs qui n'appartiennent pas à une solution de la contrainte considérée. De nombreux algorithmes ont été proposés pour assurer la consistance de domaine, tels que AC3, AC4, AC6, AC7 et AC2001 (voir [2]). Les algorithmes de consistance utilisent généralement la notion de *support*. Pour une contrainte binaire $c(x, y)$, un support pour une paire (x, a), où a est une valeur possible pour x, est une paire (y, b) tel que $c(a, b)$ est vraie. La complexité temporelle optimale pour assurer la consistance de domaine pour un CSP est de $O(ed^2)$ pour des contraintes binaires et $O(e.r.d^r)$ pour des contraintes non binaires (d est la taille du plus grand domaine, e le nombre de contraintes et r la plus grande arité des contraintes). Un algorithme tel que AC4 maintient tous les supports pour tous les couples (x, a), alors que d'autres algorithmes (par exemple, AC6) maintiennent un support unique et cherchent les supports suivants à la demande. L'algorithme AC4 fonctionne en deux étapes. Tout d'abord, il calcule tous les supports pour les paires variable/valeur de chaque contrainte. Puis, il propage la suppression d'une valeur a du domaine d'une variable x. Une propriété intéressante de l'étape de propagation est que sa complexité temporelle est proportionnel au nombre total de supports.

Example 1 *Considérons la contrainte* $x = y \bmod 10$, *avec* $D(x) = \{0..9\}$ *et* $D(y) = \{0..99\}$. *La taille de l'ensmeble des supports pour* x *est linéaire (en* $O(\#D(y))$, *où* $\#A$ *est la taille de* A*). L'étape de propagation deAC4 pour cette contrainte est aussi linéaire, alors qu'elle reste quadratique pour d'autres algorithmes AC optimaux, tels que AC6, AC7, ou AC2001.*

L'étape d'initialisation de AC4 calculant les supports est de complexité $O(d^2)$, même si le nombre de supports est de $O(d)$, car l'algorithme n'a pas connaissance de la sémantique de la contrainte. L'algorithme générique AC5 [16] a été conçu pour exploiter la sémantique des contraintes qui peut ensuite être utilisée pour produire les supports de ces contraintes en temps linéaire. Ceci permet d'obtenir un algorithme de consistance de domaine en $O(ed)$.

1. Cet article est une version française de [8].

La question scientifique abordée dans cet article est la suivante. *Est-il possible de concevoir un algorithme de domaine consistance de complexité temporelle $O(ed)$ lorsque le nombre de supports est quadratique, mais le nombre de valeurs interdites (également appelé ensemble des conflits) est linéaire ?*

Example 2 *Considérons la contrainte $x \neq y$* mod 10*, avec $D(x) = \{0..9\}$ et $D(y) = \{0..99\}$. La taille de l'ensemble des supports pour x est de 900, d'où une complexité $O(\#D(x).\#D(y))$ dans l'étape de propagation de AC4. L'utilisation de AC3, AC7 ou AC2001 ne contribue pas à réduire cette complexité. Toutefois, la taille de valeurs interdites est de 100 ($O(\#D(y))$). Peut-on utiliser un algorithme de type AC4 pour maintenir la liste des valeurs interdites à la place des supports pour obtenir un algorithme de complexity $O(ed)$?*

Cet article répond à cette question de façon positive et apporte les contributions suivantes.

- Il propose l'algorithme NAC4 (AC4 négatif) qui réalise la consistance de domaine avce une complexity optimale de (ed^2) pour les CSP binaires. Cet algorithme maintient dynamiquement l'ensemble des valeurs interdites au lieu de maintenir les supports. Nous montrons que AC4 et NAC4 sont deux instances de l'algorithme AC5 générique. Ils peuvent être combinés naturellement dans un solveur de contraintes. AC5 peut exploiter la sémantique des contraintes pour obtenir une plus grande efficacité.
- Il identifie les catégories de contraintes pour lesquelles la consistance de domaine peut être obtenue en un temps linéaire.
- Il montre comment la combinaison AC4/NAC4 peut assurer la consistance de domaine sur des combinaisons logiques de contraintes portant sur les mêmes variables.
- Il montre que la combinaison de AC4/NAC4 peut être facilement étendue afin de permettre l'évaluation de la validité et de l'implication d'une contrainte.
- Il présente des résultats expérimentaux montrant les avantages de la combinaison AC4/NAC4.

Travaux reliés L'idée d'utiliser des valeurs interdites n'est pas nouvelle en CP. La nouveauté de ce papier est que NAC4 maintient l'ensemble des valeurs interdites dynamiquement lors de la propagation. Dans [3, 12, 7], les auteurs utilisent des contraintes tables négatives, où le tableau décrit l'ensemble de tuples interdits. La table négative est utilisée pour trouver le prochain support d'une variable/valeur au moyen d'une recherche binaire. La table est statique, et n'est pas mise à jour lors de la propagation. Lecoutre [5] a montré que (x, a) a un support pour une contrainte $c(x, y)$ si la taille de $D(y)$ est strictement supérieure à la taille de l'ensemble initial des conflits de (x, a). Cette idée est intégrée dans un algorithme de propagation. Une fois de plus, la taille de l'ensemble des conflits n'est pas mise à jour pendant le calcul. La même idée est également proposée comme condition de support dans [14].

La consistance d'un ensemble de contraintes a été traitée de différentes manières. Certaines approches assurent la consistance de domaine, qui est NP-difficile en général. Un algorithme de domaine consistance, fondée sur AC7, a été proposée dans [4] pour la conjonction de contraintes. Lhomme [13] décrit un algorithme de domaine consistance pour toute combinaison de contraintes. Il se concentre principalement sur des contraintes données en extension. Les tuples interdits sont utilisés ici aussi à travers une table négative statique. D'autres approches calculent une approximation de consistance de domaine, comme dans [15] (cardinalité), [17] (disjonction constructive), ou [1] qui propose une algèbre pour combiner des contraintes.

2 AC5

Cette section revisite l'algorithme générique AC5 [16] en le généralisant légèrement afin de couvrir AC4, AC6 et AC2001.

Definition 1 (CSP) *Un CSP binaire $(X, D(X), C)$ est composé d'un ensmeble de n* variables $X = \{x_1, \ldots, x_n\}$*, un ensmeble de* domaines $D(X) = \{D(x_1), \ldots, D(x_n)\}$ *où $D(x)$ est l'ensemble des* valeurs *possibles pour la variable x et un ensemble de* constraintes *binaires $C = \{c_1, \ldots, c_e\}$, avec $Vars(c_i) \subseteq X$ $(1 \leq i \leq e)$. On note $d = max_{1 \leq i \leq n}(\#D(x))$.*

Soit c une contrainte avec $Vars(c) = \{x, y\}$, $a \in D(x), b \in D(y)$. $c(x/a, y/b)$ ou $c(y/b, x/a)$ désigne la contrainte où les variables x et y ont été remplacées par les valeurs a et b. Nous supposons que tester $c(x/a, y/b)$ prend un temps $O(1)$. Si $c(x/a, y/b)$ est vrai, alors $(x/a, y/b)$ est appelé un support de c et (y, b) est un support de (x, a) pour c. Si $c(x/a, y/b)$ est faux, alors $(x/a, y/b)$ est appelé un conflit de c et (y, b) est un conflit (ou valeur interdite) de (x, a) pour c.

Definition 2 *Soit c une contrainte avec $Vars(c) = \{x, y\}$. L'ensemble des valeurs inconsistantes, consistantes et valides pour x dans c par rapport à un en-*

semble de valeurs B sont définis comme suit.

$$Inc(c,x,B) = \{(x,a) |\ a \in D(x) \wedge \forall b \in B : \neg c(x/a, y/b)\}$$
$$Cons(c,x,B) = \{(x,a) |\ a \in D(x) \wedge \exists b \in B : c(x/a, y/b)\}$$
$$Valid(c,x,B) = \{(x,a) |\ a \in D(x) \wedge \forall b \in B : c(x/a, y/b)\}$$

Nous utilisons $Inc(c,x)$ pour denoter $Inc(c,x,D(y))$, et de même pour les autres ensembles.

Definition 3 (Domaine Consistance) *Une contrainte c sur $\{x,y\}$ est* domaine consistante *par rapport à $D(X)$ ssi $Inc(c,x) = \emptyset$ et $Inc(c,y) = \emptyset$. Un CSP $(X, D(X), C)$ est* domaine consistant *ssi toutes ses contraintes sont domaine consistantes par rapport à $D(X)$.*

La Spécification 1 décrit les méthodes principales utilisées par AC5. L'algorithme utilise une file Q de triplets (c,x,a) indiquant que la domaine consistance de la contrainte c devra être reconsidérée car la valeur a a été retirée de $D(x)$. Quand une valeur d'un domaine est supprimée, la méthode `enqueue` met les informations nécessaires dans la file. Dans la postcondition, Q_0 représente la valeur de Q lors de l'appel. Le paramètre $C1$ nous permet de considérer un sous-ensemble de contraintes, nécessaire à l'initialisation. Tant que la file d'attente contient un triplet (c,x,a), il est souhaitable que l'algorithme puisse considérer que, selon la perspective de la contraine c, la valeur a est toujours dans $D(x)$. Ceci est formalisé ci-après.

Definition 4 *La vue locale d'un domaine $D(x)$ par rapport à une file Q pour une contrainte c est l'ensemble $D(x,Q,C) = D(x) \cup \{a | (c,x,a) \in Q\}$.*

Example 3 *Étant donné une file $Q = \{(c_1,y,2),(c_1,z,2),(c_2,y,3)\}$ et domaines $D(x) = \{1,2\}$, $D(y) = D(z) = \{1\}$, on a $D(x,Q,c_1) = D(y,Q,c_1) = D(z,Q,c_1) = \{1,2\}$.*

La méthode principale de l'algorithme AC5 est `valRemove`, où l'ensemble $\triangle$ (appelé ensemble delta dans le folklore de CP en raison de l'utilisation de la lettre $\triangle$ dans la description originale d'AC5) est l'ensemble des valeurs qui ne sont plus prises en charge en raison de la suppression de la valeur b dans $D(y)$. Dans cette spécification, b est une valeur qui n'est plus dans $D(y)$ et `valRemove` calcule les valeurs de (x,a) qui ne sont plus supportées en raison de la suppression de b de $D(y)$. Notons que les valeurs dans la file (pour la variable y) sont toujours considérées comme des supports potentiels etant donné que celles-ci n'ont pas encore été prises en compte dans cette contrainte. Nous limitons également notre attention aux valeurs qui ont b dans leur supports (par exemple, $(x,a) \in Cons(c,x,\{b\})$). Cependant, nous laissons à `valRemove` la possibilité de réaliser plus d'élagage ($\triangle_2$), ce qui est utile pour les contraintes monotones [16].

```
1  enqueue(in x: Variable;in a: Value;
2          in C1: Set of Constraints;
3          inout Q: Queue)
4  // Pre: x ∈ X, a ∉ D(x) et C1 ⊆ C
5  // Post: Q = Q0 ∪ {(c,x,a)|c ∈ C1, x ∈ Vars(c)}
6
7  post(in c: Constraint;out △: Set of Values)
8  // Pre: c ∈ C with Vars(c) = {x,y}
9  // Post: △ = Inc(c,x) ∪ Inc(c,y)
10 //   + initialisation de structures de données spécifiques
11
12 boolean valRemove(in c: Constraint;
13                   in y: Variable; in b: Value;
14                   out △: Set of Values)
15 // Pre: c ∈ C, Vars(c) = {x,y} , b ∉ D(y,Q,c)
16 // Post:  △1 ⊆ △ ⊆ △2
17 //    avec  △1 = Inc(c,x,D(y,Q,c)) ∩ Cons(c,x,{b})
18 //    et    △2 = Inc(c,x)
```

Specification 1 – Méthodes enqueue, post et valRemove pour AC5.

L'algorithme AC5 est décrit dans l'algorithme 1. La fonction `propagateQueueAC5` applique `valRemove` sur chaque élément de la file jusqu'à ce que la file soit vide. La fonction `initAC5` initialise la file. La Fonction `post(c, △)` calcule les valeurs inconsistantes de la contrainte c. Si elle supprime des valeurs de domaines, seules les contraintes déjà postées sont considérées par les appels de `enqueue`. Les contraintes qui ne sont pas encore postées ne sont pas concernées par cette suppression étant donné qu'elles vont utiliser le nouveau domaine des variables lors du post. L'appel de `post` généralement initialise certaines structures de données à utiliser dans `valRemove`. Avec une légère généralisation de la spécification de `post` et de `valRemove`, l'algorithme AC5 gère également les contraintes non binaires. AC5 est générique ; la mise en oeuvre de `post` et de `valRemove` est libre. Différentes contraintes peuvent avoir leur propre implémentation de ces fonctions. Cela permet a AC5 de combiner, dans un cadre unique, différents algorithmes tels que AC4, AC6, AC7 et AC2001 et d'exploiter la sémantique des contraintes pour atteindre une meilleure efficacité.

Proposition 1 *En supposant une implémentation correcte de* `post` *et de* `valRemove`*, AC5 est correct par rapport à sa spécification.*

Dans AC5, un élément (c,x,a) ne peut être mis qu'une seule fois dans la file. La taille de la file donc de $O(e.r.d)$ pour les CSP non binaires et $O(ed)$ pour les CSP binaires. Le nombre d'exécutions de `valRemove` est aussi borné par $O(e.r.d)$.

Proposition 2 *Pour les CSP binaires, si la com-*

```
1  AC5(in X, C, inout D(X)){
2  // Pre: (X, D(X), C) is a CSP
3  // Post: D(X) ⊆ D(X)0, (X, D(X), C) equivalent to
4  //       (X, D(X)0, C)
5  //       et (X, D(X), C) is domain consistent
6    initAC5(Q);
7    propagateQueueAC5(Q);
8  }

9  initAC5(out Q){
10   Q = ∅;
11   C1 = ∅;
12   forall(c in C){
13     C1 += c;
14     post(c,△);
15     forall((x, a) in △){
16       D(x) -= a;
17       enqueue(x, a,C1,Q);
18     }
19   }
20 }

21 propagateQueueAC5(in Q){
22   while Q != ∅ {
23     select( (c, y, b) in Q) {
24       Q = Q - (c, y, b);
25       valRemove(c, y, b,△);
26       forall((x, a) in △){
27         D(x) -= a;
28         enqueue(x, a,C,Q);
29       }
30     }
31   }
32 }
```

Algorithm 1 – Algorithme AC5.

plexité temporelle de `post` *est de* $O(d^2)$ *et la complexité de* `valRemove` *est de* $O(d)$*, alors la complexité temporelle de AC5 est optimale et est de* $O(e.d^2)$*. Si la complexité temporelle de* `post` *est de* $O(d)$ *et la complexité amortie de toutes les exécutions de* `valRemove` *pour une contrainte est de* $O(d)$ *(par exemple, la complexité temporelle de* `valRemove` *est* $O(\triangle)$*), alors la complexité spatiale et temporelle de AC5 est* $O(ed)$*.*

Nous présentons l'algorithme AC4 comme une instanciation de AC5 en donnant le code des méthodes `post` et `valRemove` (Algorithmes 2). La méthode `valRemoveAC4` utilise une structure de données S pour enregistrer les supports de chaque valeur dans les différentes contraintes. Elle est initialisée par `postAC4` et satisfait l'invariant suivant à la ligne 21 de l'algorithme 1 (AC5).

```
1  postAC4(in c: Constraint;out △: Set of Values) {
2  // Pre: c ∈ C with Vars(c) = {x, y}
3  // Post: △ = Inc(c, x) ∪ Inc(c, y)
4  //       + initialisation de la structure de données S
5    post_varAC4(c, x, △1);
6    post_varAC4(c, y, △2);
7    △ = △1 ∪ △2;
8  }
9  post_varAC4(in c: Constraint;in x: Variable;
10                out △: Set of Values) {
11   △ = ∅;
12   forall(a in D(x)){
13     S[x, a, c] = ∅;
14     forall(b in D(y) : c(x/a, y/b))
15       S[x, a, c] += b ;
16     if (S[x, a, c]==∅)
17       △ += (x, a) ;
18   }
19 }
20 valRemoveAC4(in c: Constraint;in y: Variable;
21            in b: Value; out △: Set of Values) {
22 // Pre: c ∈ C, Vars(c) = {x, y} , b ∉ D(y, Q, c)
23 // Post: △ = Inc(c, x, D(y, Q, c)) ∩ Cons(c, x, {b})
24   △ = ∅;
25   forall(a in S[y, b, c]) {
26     S[x, a, c] -= b ;
27     if (S[x, a, c]==∅ & a in D(x))
28       △ += (x, a) ;
29   }
30 }
```

Algorithm 2 – Méthodes `post` et `valRemove` de AC4.

Soit $c \in C$ avec $Vars(c) = \{x, y\}$:

$(1.x)$ $\forall a \in D(x, Q, c))$:
$S[x, a, c] = \{b \in D(y, Q, c) | c(x/a, y/b)\}$
$(2.x)$ $\forall a \in D(x) : S[x, a, c] \neq \emptyset$

Et de manière similaire pour y. Cet invariant assure l'exactitude de `valRemoveAC4`. Après l'appel de `postAC4`, nous avons $\sum_{a \in D(x)} \#S[x, a, c] = \sum_{b \in D(y)} \#S[y, b, c]$ qui est $O(d^2)$. La taille de la structure de donnée est de $O(e.d^2)$.

3 NAC4

NAC4 (AC4 Négatif) est une nouvelle instance de AC5 qui est basée sur les valeurs interdites. Celles-ci sont dynamiquement maintenues lors de la propagation. Par NAC4, nous désignons l'algorithme AC5 dont les méthodes `postNAC4` et `valRemoveNAC4` sont définies comme décrites dans les Algorithmes 3 et 4.

NAC4 utilise une structure de données F pour enregis-

```
1  postNAC4(in c: Constraint;out △: Set of Values) {
2  // Pre: c ∈ C with Vars(c) = {x, y}
3  // Post: △ = Inc(c, x) ∪ Inc(c, y)
4  //         + initialisation des structures de données F,
5  //           setOfSize et localSize
6    post_varNAC4(c,x,△1);
7    post_varNAC4(c,y,△2);
8    localSize[x, c] = #D(x);
9    localSize[y, c] = #D(y);
10   △ = △1 ∪ △2;
11 }
12 post_varNAC4(in c: Constraint;in x: Variable;
13               out △: Set of Values) {
14   △ = ∅;
15   forall(k in 0..#D(y))
16     setOfSize[x, k, c] = ∅;
17   forall(a in D(x)){
18     F[x, a, c] = ∅;
19     forall(b in D(y) : ¬c(x/a, y/b))
20       F[x, a, c] += b ;
21     k = #F[x, a, c];
22     setOfSize[x, k, c]) += a;
23     if (k==#D(y))
24       △ += (x, a) ;
25   }
26 }
```

Algorithm 3 – La méthode `post` de NAC4.

```
1  valRemoveNAC4(in c: Constraint;in y: Variable;
2            in b: Value,out △: Set of Values) {
3  // Pre: c ∈ C, Vars(c) = {x, y} , b ∉ D(y, Q, c)
4  // Post: △ = Inc(c, x, D(y, Q, c)) ∩ Cons(c, x, {b})
5    localSize[y, c]-- ;
6    k = #F[y, b, c];
7    setOfSize[y, k, c] -= b;
8    forall(a in F[y, b, c]){
9      F[x, a, c] -= b ;
10     k = #F[x, a, c];
11     setOfSize[x, k + 1, c] -= a;
12     setOfSize[x, k, c] += a;
13   }
14   △ = ∅;
15   s = localSize[y, c];
16   forall(a in setOfSize[x, s, c] : a in D(x))
17     △ += (x, a);
18 }
```

Algorithm 4 – La méthode valRemove de NAC4.

trer les valeurs interdites pour chaque valeur dans les différentes contraintes. Pour une contrainte c, définie sur x, y, l'idée de base est que la valeur a devrait être retirée de $D(x)$ dès que l'ensemble des valeurs interdites pour (x, a) et l'ensemble $D(y)$ sont les mêmes. Ce test peut être réalisé efficacement (1) en raisonnant sur la taille de l'ensemble des valeurs interdites pour (x, a) et de l'ensemble $D(y)$, (2) en utilisant une structure de données triant par taille les ensmebles de valeurs interdites, et (3) en enregistrant la taille de la vue locale des domaines. Ces structures de données sont initialisées dans `postNAC4` et mises à jour dans `valRemoveNAC4`. La structure de données $F[x, a, c]$ enregistre l'ensemble des valeurs interdites pour (x, a) et c; $setOfSize[x, k, c]$ désigne l'ensemble des valeurs b telles que $\#F[x, b, c] = k$ et $localSize[x, c]$ désigne la taille de la vue locale du domaine $D(x)$. La méthode `valRemoveNAC4` met tout d'abord à jour la taille de la vue locale de $D(y)$ et supprime b de la structure de données $setOfSize$ pour la variable y (lignes 5 à 7). Il met ensuite à jour l'ensemble des valeurs interdites ainsi que $setOfSize$ pour chaque paire $(x, a) \in F[y, b, c]$ (lignes 8 à 13). Enfin, il supprime les valeurs qui ne sont plus supportées, c'est-à-dire, les valeurs de $setOfSize[x, s, c] \cap D(x)$, où s est la taille locale de $D(y)$ (lignes 14 à 18).

Les structures de données satisfont l'invariant suivant à la ligne 21 de l'Algorithme 1 (AC5). Soit $c \in C$ avec $Vars(c) = \{x, y\}$:

$$
\begin{array}{ll}
(3.x) & \forall a \in D(x, Q, c) : \\
 & F[x, a, c] = \{b \in D(y, Q, c) | \neg c(x/a, y/b)\} \\
(4.x) & \forall a \in D(x) : F[x, a, c] \subset D(y, Q, c) \\
(5.x) & setOfSize[x, k, c] = \{a \in D(x, Q, c))| \\
 & \#F[x, a, c] = k\}(0 \leq k \leq \#D(y, Q, c)) \\
(6.x) & localSize[x, c] = \#D(x, Q, c)
\end{array}
$$

et de manière similaire pour y. A partir de ces invariants, nous avons que $F[x, a, c] \subseteq D(y, Q, c)$ est vrai à la ligne 24 et que la valeur a doit être supprimée de $D(x)$ si $F[x, a, c] = D(y, Q, c)$. Donc, si $s = localSize[y, c]$, l'algorithme doit retirer les valeurs de $setOfSize[x, s, c]$ du domaine de $D(x)$. Ces invariants assurent l'exactitude de `valRemoveNAC4`.

La taille des structure de données est de $O(e.d^2)$. Dans l'élagage de la méthode postNAC4, la vue locale de la taille du domaine $D(y)$ est $\#D(y)$ vu que la file ne contient pas d'éléments de la forme $(x, ., c)$. La complexité de `postNAC4` est $O(d^2)$ et de la complexité de `valRemoveNAC4` est de $O(d)$. Ainsi, par la propriété 2, la complexité globale de NAC4 est de $O(ed^2)$, qui est la complexité optimale pour assurer la consistance de domaine sur les CSP binaires.

Example 4 *NAC4 est illustré sur le CSP suivant :* $c_1(x, y) = \{(1, 4), (1, 5), (2, 2), (2, 5), (3, 1), (3, 3), (3, 4)\}$, $c_2 : y \neq 4$, $c_3 : y \neq 5$, $D(x) = \{1, 2, 3\}$ *et* $D(y) = \{1, 2, 3, 4, 5\}$. *L'execution de* `postNAC4`$(c_1, \triangle)$ *produit* $\triangle = \emptyset$ *et remplit les structures de données*

comme suit :

$F[x,1,c_1]=\{1,2,3\}$ $F[y,1,c_1]=\{1,2\}$
$F[x,2,c_1]=\{1,3,4\}$ $F[y,2,c_1]=\{1,3\}$
$F[x,3,c_1]=\{2,5\}$ $F[y,3,c_1]=\{1,2\}$
$F[y,4,c_1]=\{2\}$
$F[y,5,c_1]=\{3\}$

$setOfSize[x,1,c_1]=\emptyset$ $setOfSize[y,1,c_1]=\{4,5\}$
$setOfSize[x,2,c_1]=\{3\}$ $setOfSize[y,2,c_1]=\{1,2,3\}$
$setOfSize[x,3,c_1]=\{1,2\}$ $setOfSize[y,3,c_1]=\emptyset$
$setOfSize[x,4,c_1]=\emptyset$
$setOfSize[x,5,c_1]=\emptyset$
$localSize[x,c_1] = 3$ $localSize[y,c_1] = 5$

`postNAC4`$(c_2,\triangle)$ *retourne* $\triangle = \{(y,4)\}$ *et* `postNAC4`$(c_3,\triangle)$ *produit* $\triangle=\{(y,5)\}$*, ce qui donne* $Q=\{(c_1,y,4),(c_1,y,5)\}$*,* $D(x)=\{1,2,3\}$*, et* $D(y)=\{1,2,3\}$*. La methode* `valRemoveNAC4`$(c_1,y,4,\triangle)$ *met à jour les variables suivantes :*

$$\begin{array}{lcl} F[x,2,c_1] & = & \{1,3\} \\ setOfSize[x,3,c_1] & = & \{1\} \\ setOfSize[x,2,c_1] & = & \{2,3\} \\ localSize[y,c_1] & = & 4 \\ setOfSize[y,1,c_1] & = & \{5\} \end{array}$$

Etant donné que $setOfSize[x,4,c_1]=\emptyset$*, on a* $\triangle=\emptyset$*,* $Q=\{(c_1,y,5)\}$*,* $D(x)=\{1,2,3\}$*, et* $D(y)=\{1,2,3\}$*.* `valRemoveNAC4`$(c_1,y,5,\triangle)$ *met à jour les variables suivantes :*

$$\begin{array}{lcl} F[x,3,c_1] & = & \{2\} \\ setOfSize[x,2,c_1] & = & \{2\} \\ setOfSize[x,1,c_1] & = & \{3\} \\ localSize[y,c_1] & = & 3 \\ setOfSize[y,1,c_1] & = & \emptyset \end{array}$$

Etant donné que $setOfSize[x,3,c_1]=\{1\}$*, on a* $\triangle=\{(x,1)\}$*,* $Q=\{(c_1,x,1)\}$*,* $D(x)=\{2,3\}$*, et* $D(y)=\{1,2,3\}$*.* `valRemoveNAC4`$(c_1,x,1,\triangle)$ *met à jour les variables suivantes :*

$$\begin{array}{lcl} F[y,1,c_1] & = & \{2\} \\ F[y,2,c_1] & = & \{3\} \\ F[y,3,c_1] & = & \{2\} \\ setOfSize[y,2,c_1] & = & \emptyset \\ setOfSize[y,1,c_1] & = & \{1,2,3\} \\ localSize[x,c_1] & = & 2 \\ setOfSize[x,3,c_1] & = & \emptyset \end{array}$$

Les domaines deviennent finalement $D(x)=\{2,3\}$ *et* $D(y)=\{1,2,3\}$*.*

Proposition 3 *Soit* $c \in C$ *sur les variables* $\{x,y\}$*. Les Invariants (3-6.x-y) sont satisfaits a la ligne 21 de AC5.*

Proposition 4 *NAC4 est correct et sa complexité spaciale et temporelle est de* $O(e.d^2)$*.*

4 Applications

Nous allons maintenant examiner une série d'applications liées au principe du maintien dynamique des valeurs interdites.

Contraintes AC éparses Les instances AC4 et NAC4 de AC5 peuvent être combinées ; chaque contrainte mettant en oeuvre sa version AC4 ou NAC4 des méthodes `post` et `valRemove` valRemove. Ceci sera noté AC5(AC4,NAC4). Une propriété intéressante de AC5(AC4,NAC4) est que la complexité amortie de toutes les exécutions de `valRemoveAC4` ou `valRemoveNAC4` pour une contrainte est limitée par le nombre d'éléments dans la structure de données S ou F de cette contrainte. On obtient alors la spécialisation suivante de la proposition 2.

Proposition 5 *Si une version spécialisée de* `postAC4` *ou* `postNAC4` *exploitant la sémantique de la contrainte s'exécute en un temps* $O(K)$ *pour chaque contrainte d'un CSP binaire, alors la complexité spaciale et temporelle de AC5(AC4,NAC4) est* $O(e.K)$*.*

Comme cas particulier, si S ou F peut être rempli en $O(d)$, un algorithme de domaine consistance s'exécute en un temps $O(ed)$, tel que formalisé par la classe contraintes suivante.

Definition 5 *Une contrainte c avec* $Vars(c)=\{x,y\}$ *est* positivement éparse *par rapport à un domaine D si et seulement si* $\#\{(a,b)\in D^2 | c(x/a,y/b)\}$ *est* $O(\#D)$*. La contrainte c est* négativement éparse *par rapport à un domaine D si et seulement si* $\neg c$ *est* positivement éparse *par rapport D.*

Example 5 *Des exemples de contraintes positivement et négativement éparses sont les contraintes bijectives* ($x+y=k$*, où k est une constante), les contraintes anti-bijectives* ($x+y\neq k$*), les contraintes fonctionnelles* ($x=|y-k|$ *ou* $x=y \bmod k$*), les contraintes anti-fonctionnelles* ($x\neq|y-k|$ *ou* $x\neq y \bmod k$*), mais comprennent aussi des des contraintes (anti-)fonctionnelles telles que*$|x-y|=k$ *et* $|x-y|\neq k$*. On peut aussi considérer les contraintes de congruence, tels que* $(x+y) \bmod k = 0$ *et* $(x+y) \bmod k \neq 0$ *qui sont éparses lorsque k est* $O(d)$*.*

Grâce à la généricité d'AC5, nous pouvons exploiter la sémantique des contraintes spécifiques dans la méthode `postAC4` des contraintes positivement éparses et de `postNAC4` pour les contraintes négativement éparses afin de remplir la structure de données S ou F en $O(d)$ et obtenir ainsi une complexité temporelle de $O(d)$.

Proposition 6 *Pour des contraintes positivement et*

négativement éparses, la complexité spaciale et temporelle est de $O(ed)$.

Combinaison de constraintes sur des mêmes variables Considérons maintenant une contrainte c sur $\{x, y\}$ définie comme une combinaison booléenne de contraintes $\{c_1, \ldots, c_k\}$ portant sur les mêmes variables. Supposons pour simplifier que le nombre de connecteurs logiques est bornée par k. La contrainte c peut être postée en AC5(AC4,NAC4) avec une complexité de $O(k.d^2)$. L'étape de propagation sur cette contrainte sera alors réalisée en temps $O(K)$ pour atteindre la consistance de domaine, où K est le nombre de supports.

Example 6 *Considérons la contrainte $c \equiv (c_1 \wedge c_2) \vee (c_3 \wedge c_4)$ où $c_1 \equiv x \neq |y-2|$, $c_2 \equiv y - 1 \neq x \bmod 2$, $c_3 \equiv x = |y-1|$, $c_4 \equiv |x-2| = y$, avec $D(x) = \{0, 1\}$ et $D(y) = \{1, 2\}$. Chaque contrainte c_i est domaine consistante, mais ni $c_1 \wedge c_2$ ni $c_3 \wedge c_4$ ne l'est. Une application de AC4 (ou NAC4) sur c permettra de détecter une inconsistance.*

Dans certains cas, comme dans l'exemple ci-dessus, il est possible de parvenir à une meilleure complexité en exploitant conjointement les supports et les valeurs interdites. L'idée centrale est que chaque contrainte c_i doit utiliser les supports ou les valeurs interdites en fonction de sa sémantique. Ensuite, les contraintes individuelles sont combinées par des opérateurs logiques qui utilisent les supports et les valeurs interdites afin de calculer ses propres supports ou valeurs interdites, et cela de façon récursive. Le Tableau 1 présente les règles permettant de combiner des contraintes et de calculer la structure de données S ou F pour les variables x et y, selon la structure de données maintenue dans les sous-expressions. Les règles sont données pour la variable x, mais sont similaires pour y. Une contrainte c_i qui utilise la structure de données S (resp. F) sera notée c_i^+ (resp. c_i^-). Si la méthode post applique ces règles à la contrainte c, alors l'algorithme réalise ensuite la consistance de domaine. Il n'y a pas coût temporel ou spatial supplémentaire étant donné que toutes les opérations du tableau 1 peuvent être effectuées en temps $s_1 + s_2$, où s_i est la taille de la structure de données (S ou F) pour c_i. Comme cas particulier, si la complexité temporelle pour poster chaque c_i est de $O(d)$ (e.g. contrainte fonctionnelle ou anti-fonctionnelle), la complexité spatiale et temporelle pour poster la contrainte c est de $O(k.d)$.

Proposition 7 *Étant donné un ensemble de contraintes binaires C et une contrainte binaire c exprimée comme une combinaison logique des contraintes $c_1, \ldots, c_k$ avec $Vars(c) = Vars(c_i)$ $(1 \leq i \leq k)$, si la méthode* `post` *pour c applique les règles de la table 1, alors l'algorithm AC5(AC4,NAC4) sur $C \cup \{c_1, \ldots, c_k\}$ réalise la domaine consistance. Si la complexité temporelle des méthodes* `post` *des contraintes de $C \cup \{c_1, \ldots, c_k\}$ est de $O(d)$, alors la complexité temporelle et spatiale de AC5(AC4,NAC4) appliqué sur $C \cup \{c\}$ est $O((e+k).d)$, avec $e = \#C$.*

$c^- \equiv \neg c_1^+$	$F[x,a,c] = S[x,a,c_1]$
$c^+ \equiv \neg c_1^-$	$S[x,a,c] = F[x,a,c_1]$
$c^+ \equiv c_1^+ \wedge c_2^+$	$S[x,a,c] = S[x,a,c_1] \cap S[x,a,c_2]$
$c^- \equiv c_1^- \wedge c_2^-$	$F[x,a,c] = F[x,a,c_1] \cup F[x,a,c_2]$
$c^+ \equiv c_1^+ \wedge c_2^-$	$S[x,a,c] = S[x,a,c_1] \setminus F[x,a,c_2]$
$c^+ \equiv c_1^+ \vee c_2^+$	$S[x,a,c] = S[x,a,c_1] \cup S[x,a,c_2]$
$c^- \equiv c_1^- \vee c_2^-$	$F[x,a,c] = F[x,a,c_1] \cap F[x,a,c_2]$
$c^- \equiv c_1^+ \vee c_2^-$	$F[x,a,c] = F[x,a,c_2] \setminus S[x,a,c_1]$

TABLE 1 – Règles pour combiner $c_1(x, y)$ et $c_2(x, y)$.

Validité et implication AC5(AC4,NAC4) peut être étendu pour supporter les méthodes `isValid` et `isEntailed` (Spécifications 2). Avec AC4, une variable/valeur (x, a) est détectée comme étant valide dans c si la taille de $S[x, a, c]$ est $\#D(y, Q, c)$. AC4 doit pour cela maintenir les structures $setOfSize$ et $localSize$ structures de données de NAC4. Avec NAC4, une variable/valeur (x, a) est détectée comme étant valide dans c si $F[x, a, c]$ est vide. L'invariant de la structure de données pour AC4 et NAC4 serait alors (1-6.x-y). La complexité théorique de AC5(AC4,NAC4) est inchangée, tandis que la complexité pratique est à peu près doublée. AC4 et NAC4 garderaient ainsi le nombre de valeurs valides pour chaque contrainte c. Si les valeurs valides pour x dans c atteint $\#D(x, Q, C)$, alors la contrainte est détectés toujours vraie (en supposant que les domaines soient non vides). On pourrait aussi étendre facilement les méthodes `post` et `valRemove` à des méthodes `post`(c,$\triangle^-$,$\triangle^+$) et `valRemove`(c,y,b,$\triangle^-$,$\triangle^+$) où l'argument supplémentaire $\triangle^+$ retourne l'ensemble des nouvelles valeurs valides, défini comme

$$\triangle^+ = Valid(c, x) \cup Valid(c, y)$$

pour `post`, et

$$\triangle^+ = Valid(c, x, D(y, Q, c)) \cap Inc(c, y, \{b\})$$

pour `valRemove`. Ces algorithmes étendus de domaine consistance sont utiles pour des combinateurs de contraintes, la réification ainsi que dans un cadre *Ask & Tell*.

```
1  Boolean isValid(in c:  Constraint,
2                  in x: Variable, in a: Value)
3  // Pre: c ∈ C, Vars(c) = {x, y}, a ∈ D(x),
4  //      D(y) ≠ ∅
5  // Post: return true iff (x, a) ∈ Valid(c, x, D(y, Q, c))
6  Boolean isEntailed(in c:  Constraint)
7  // Pre: c ∈ C with Vars(c) = {x, y},
8  //      D(x) ≠ ∅, D(y) ≠ ∅
9  // Post: return true iff
10 //         ∀a ∈ D(x) : (x, a) ∈ Valid(c, x, D(y, Q, c))
```

Specification 2 – Les méthodes `isValid` et `isEntailed`.

Combinaison de contraintes sur des variables différentes Assurer la consistance de domaine sur une combinaison de contraintes (binaires) portant sur des variables différentes est un problème NP-difficile. Une approximation de consistance de domaine peut être réalisée en utilisant le cadre proposé dans [1], où les contraintes primitives produisent non seulement des valeurs incompatibles, mais aussi les valeurs valides. Notre algorithme AC5(AC4,NAC4) étendu peut être utilisé pour combiner les contraintes en utilisant l'algèbre proposée dans [1]

5 Résultats experimentaux

Cette section illustre les avantages d'une exploitation conjointe des supports et des valeurs interdites. Nous avons évalué AC4, NAC4, et leur combinaison sur des CSPs impliquant lees contraintes positivement et négativement éparses $x = y \bmod k$, $x = |y-k|$, $x+y = k$, $|x - y| = k$, $(x + y) \bmod k = 0$ et leurs versions négatives, où k est une constante. Trois séries de 20 CSPs ont été générés. Le premier ensemble contient uniquement des contraintes positives (CPOS), le second uniquement des contraintes négatives (CNEG), et le troisième des contraintes positives et négatives (cPosNeg). Les résultats sont présentés dans la Table 2. Le nom de Cneg_50_200_10 signifie que chaque CSP a 50 variables avec un domaine {0..199}, et qu'il y a 10% de contraintes entre toutes les paires de variables distinctes. Les paramètres ont été choisis pour éviter les CSPs trivialement satisfaisables ou trivialement inconsistants. Les contraintes ont été choisies au hasard en utilisant une distribution uniforme. Les valeurs k sont également déterminées en utilisant une distribution uniforme. Chaque CSP a été résolu en Comet [9] en utilisant quatre algorithmes de consistance différents : (1) AC4 pour chaque contrainte, (2) NAC4 pour chaque contrainte, (3) la combinaison AC5(AC4,NAC4) en utilisant AC4 pour les contraintes positives et NAC4 pour les contraintes négatives et (4) la combinaison AC5(AC4*,NAC4*) qui est semblable à AC5(AC4,NAC4), mais utilise des méthodes spécialisées `post` (linéaires) exploitant la sémantique des contraintes. Pour un CSP qui ne contient que des contraintes positives, AC5(AC4,NAC4) se réduit à AC4 et, pour un CSP qui ne contient que des contraintes négatives, AC5(AC4,NAC4) se réduit à NAC4. Nous donnons le temps d'exécution moyen (en secondes), ainsi que le pourcentage de CSPs consistants dans chaque ensemble de données. L'inconsistance d'un CSP est toujours détectée dans le noeud racine de l'arbre de recherche. Pour les CSPs consistants, la recherche est arrêtée après 1000 noeuds. Les expériences ont été réalisées sur un seul coeur d'un ordinateur avec processeur Intel Core Duo cadencé à 2,8 GHz et avec 4 Go de mémoire.

Pour les contraintes positivement éparses, AC4 est beaucoup plus efficace (facteur d'accélération de 41) que NAC4, tandis que NAC4 est beaucoup plus efficace que AC4 (facteur d'accélération de 11,5) pour les contraintes négativement éparses. Ceci montre l'intérêt de NAC4. L'utilisation d'une contrainte post spécialisée conduit à un facteur d'accélération de 2,57 pour AC4 et de 1,17 pour NAC4. Pour les CSPs combinant des contraintes positivement et négativement éparses, NAC4 est 3,5 fois plus lent que AC4, ce qui s'explique par les structures de données plus complexes à maintenir pour NAC4. Cette dernière série de CSPs montre l'intérêt d'un algorithme générique permettant la combinaison de différents algorithmes tels que AC4 et NAC4. Le facteur d'accélération de AC5(AC4,NAC4) par rapport à AC4 est de 12,7. Il augmente jusqu'à 14,9 lorsque l'on utilise des méthodes post spécialisée dans AC5(AC4*,NAC4*).

6 Conclusion

Ce papier propose l'algorithme optimal NAC4 pour la domaine consistance ; cet algorithme n'utilise pas les supports, mais maintient dynamiquement les valeurs interdites lors de la propagation. Les principes de NAC4 peuvent être combinés au sein de l'algorithme générique AC5 avec les techniques utilisées dans AC4, AC6, et AC2001 pour exploiter la sémantique des contraintes et obtenir ainsi une plus grande efficacité. En particulier, les valeurs interdites permettent à AC5 d'obtenir la consistance de domaine en un temps $O(ed)$ pour les catégories de contraintes dans lesquelles le nombre de supports est de $O(d^2)$, mais le nombre de

	% Consist.	AC4	NAC4	AC5(AC4,NAC4)	AC5(AC4*,NAC4*)
cPos_10_200_01	55%	0.466	19.198	-	0.181
cNeg_50_200_10	100%	27.596	2.381	-	2.027
cPosNeg_50_200_05	53%	21.690	76.073	1.700	1.454

TABLE 2 – Comparaison de AC4, NAC4, AC5(AC4,NAC4) et AC5(AC4*/NAC4*).

valeurs interdites est de $O(d)$. Cet article montre également comment les valeurs interdites et les supports peuvent être utilisés conjointement pour assurer la consistance de domaine sur des combinaisons logiques de contraintes et de calculer la validité et l'implication de contraintes. Les résultats expérimentaux montrent que la combinaison de supports et de valeurs interdites permet de considérablement réduire le coût de calcul de la consistance de domaine pour certaines classes de contraintes.

Les travaux futurs visent à comparer AC5(AC4/NAC4) avec d'autres algorithmes de domaine consistance, à réaliser une évaluation expérimentale sur d'autres jeux de tests, y compris des instances non-binaires ainsi qu'à étendre NAC4 pour gérer des tables négativess représentées de manière compacte, telle que décrits dans [10, 11, 6, 7].

Remerciements

Cette recherche est partiellement financée par le programme des Pôles d'Attraction Interuniversitaires (Politique scientifique belge), ainsi que par le projet FRFC 2.4504.10 du Fond National belge de la Recherche Scientifique.

Références

[1] Fahiem Bacchus and Toby Walsh. Propagating logical combinations of constraints. In *IJCAI*, pages 35–40, 2005.

[2] Christian Bessiere. Constraint propagation. In Francesca Rossi, Peter van Beek, and Toby Walsh, editors, *Handbook of Constraint Programming*. Elsevier Science Inc., New York, NY, USA, 2006.

[3] Christian Bessière and Jean-Charles Régin. Arc consistency for general constraint networks : Preliminary results. In *IJCAI*, pages 398–404, 1997.

[4] Christian Bessière and Jean-Charles Régin. Local consistency on conjunctions of constraints. In *ECAI-98, proceedings Workshop on Non Binary Constraints*, pages 53–60, 1998.

[5] Frédéric Boussemart, Fred Hemery, Christophe Lecoutre, and Lakhdar Sais. Support inference for generic filtering. In *CP*, pages 721–725, 2004.

[6] Kenil C. K. Cheng and Roland H. C. Yap. Maintaining generalized arc consistency on ad hoc r-ary constraints. In *CP'08*, pages 509–523. Springer-Verlag, 2008.

[7] Kenil C. K. Cheng and Roland H. C. Yap. An mdd-based generalized arc consistency algorithm for positive and negative table constraints and some global constraints. *Constraints*, 15(2) :149–204, 2010.

[8] Yves Devilel and Pascal Van Hentenryck. Domain consistency and forbidden values. In *CP*, pages 191–205, 2010.

[9] Dynadec. Comet user manuel. In *www.dynadec.be*, 2010.

[10] Ian P. Gent, Chris Jefferson, Ian Miguel, and Peter Nightingale. Data structures for generalised arc consistency for extensional constraints. In *AAAI'07*, pages 191–197, 2007.

[11] George Katsirelos and Toby Walsh. A compression algorithm for large arity extensional constraints. In *CP'07*, pages 379–393. Springer-Verlag, 2007.

[12] Christophe Lecoutre. *Constraint Networks : Techniques and Algorithms*. ISTE/Wiley, 2009.

[13] Olivier Lhomme. Arc-consistency filtering algorithms for logical combinations of constraints. In *CPAIOR*, pages 209–224, 2004.

[14] Deepak Mehta and Marc R. C. van Dongen. Reducing checks and revisions in coarse-grained MAC algorithms. In *IJCAI*, pages 236–241, 2005.

[15] Pascal Van Hentenryck and Yves Deville. The cardinality operator : A new logical connective for constraint logic programming. In *ICLP*, pages 745–759, 1991.

[16] Pascal Van Hentenryck, Yves Deville, and Choh-Man Teng. A generic arc-consistency algorithm and its specializations. *Artif. Intell.*, 57(2-3) :291–321, 1992.

[17] Pascal Van Hentenryck, Vijay A. Saraswat, and Yves Deville. *Constraint Programming : Basics and Trends*, chapter Design, Implementation, and Evaluation of the Constraint Language cc(FD), pages 293–316. Springer, 1994.

Décomposition par paire pour l'optimisation combinatoire dans les modèles graphiques

Aurélie Favier, Simon de Givry, Andrés Legarra, Thomas Schiex
INRA Toulouse, France
{afavier,degivry,andres.legarra,tschiex}@toulouse.inra.fr

Abstract

Nous proposons une nouvelle décomposition additive des tables de probabilités qui préserve l'équivalence de la distribution jointe permettant de réduire la taille des potentiels, sans ajout de nouvelles variables. Nous formulons le problème de Most Probable Explanation (MPE) dans les réseaux probabilistes comme un problème de satisfaction de contraintes pondérées (Weighted Constraint Satisfaction Problem WCSP). Notre décomposition par paire permet de remplacer une fonction de coûts par des fonctions d'arités plus petites. Le WCSP résultant de cette décomposition est plus facile à résoudre par les techniques de l'état de l'art des WCSP. Même si tester la décomposition par paire est équivalent à tester l'indépendance de paire du réseau de croyances original, nous montrons comment le tester efficacement et l'appliquer, même avec des contraintes dures. De plus, nous inférons une information supplémentaire à partir des fonctions de coûts non binaires résultantes par projection&soustraction dans leurs fonctions binaires. Nous observons d'importantes améliorations grâce au pré-traitement avec la décompostion de paire et la projection&soustraction comparée aux solveurs actuels de l'état de l'art sur deux ensembles de problèmes difficiles.

1 Introduction

Les *modèles graphiques probabilistes* (Probabilistic Graphical Models PGM) permettent une approche générale pour les raisonnements automatiques sous incertitudes [6]. Les PGM couvrent les réseaux Bayésiens, les réseaux de Markov, mais également les formalismes déterministes comme les réseaux de contraintes. Ce papier s'intéresse au PGM discrets et à l'optimisation du problème MPE, qui consiste à maximiser le produit de fonctions positives sur un ensemble de variables discrètes. Dans la section 2, nous montrons comment, en utilisant une log-transformation, le problème MPE peut se reformuler en un WCSP, un formalisme général pour l'optimisation de contraintes qui consiste à minimiser la somme de *fonctions de coûts* positives sur un ensemble de variables dicrètes [12].

Les algorithmes exacts pour MPE (ou WCSP) sont pour la plupart basés sur l'algorithme de recherche Depth-First Branch and Bound (DFBB) ou sur des méthodes d'inférence, incluant l'élimination de variables, la jointure d'arbre et la compilation. Les méthodes d'inférence ont des problèmes de mémoire en général pour les problèmes importants et complexes (avec une largeur d'arbre importante). Les solveurs de l'état de l'art pour MPE combinent DFBB et les méthodes d'inférence à mémoire bornée (voir par exemple [13, 10]). DFBB est une méthode de recherche arborescente complète utilisant un espace mémoire linéaire. Pendant la recherche, pour les problèmes de minimisation, l'algorithme maintient un borne supérieure UB de la solution de coût minimum, en prenant le coût de la meilleure solution trouvée jusqu'alors. De plus, chaque nœud dans l'arbre de recherche est associé à une borne inférieure LB, une sous-estimation du coût minimum de la solution du sous-problème induit par le nœud courant. Si $LB \geq UB$, DFBB coupe l'espace de recherche sous le nœud courant.

Parmi les différentes techniques utilisées pour produire une bonne borne inférieure, les cohérences locales souples ont été introduites dans les WCSP. Leur complexité en temps dans le pire cas est exponentielle par rapport à la plus grande arité (nombre de variables) des fonctions de coûts [1], elles sont généralement seulement appliquées sur les fonctions de coûts de petites arités (2 ou 3). Par conséquent il est souhai-

Ce travail a bénéficié partiellement d'une aide de l'Agence Nationale de la Recherche portant la référence ANR-10-BLA-0214.

table de décomposer les fonctions de coûts en fonctions de coûts de plus petite arité. Notons que l'élimination de variables peut également bénéficier de cette décomposition (car le nombre de voisins d'une variable peut décroître).

Il est bien connu que les *indépendances conditionnelles* (IC) permettent de factoriser les distributions de probabilités. Toutefois ni les réseaux Baysiens ni les réseaux de Markov peuvent explicitement représenter toutes les IC parfaitement [6][1].

IC dépendants du contexte, telles que $(X \perp\!\!\!\perp Y \mid Z = z)$, qui ne sont pas explicites dans le réseau et qui peut être exploitée après avoir l'observation $Z = z$ ou réalisé l'élimination de variables ou éliminé des valeurs par cohérence locale souple. Dans les réseaux de Markov, plusieurs factorisations existent dans le cas des distributions non strictement positives. Dans la section 3, nous montrons comment exploiter efficacement l'indépendance de paire dans les WCSP pour améliorer DFBB avec les cohérences locales souples et l'élimination de variables.

2 Préliminaires

Un *Modèle Graphique Probabiliste* (PGM) (ou *distribution de Gibbs*) est défini par un produit de fonctions positives $\mathcal{F}$, sur un ensemble de variables discrètes $\mathcal{X}$, exprimant une information probabiliste ou déterministe [6]. Les sous-ensembles de $\mathcal{X}$ seront notés par des lettres en gras comme $\mathbf{S}$.

Définition 1 (PGM)**.** *Un PGM est un triplet $(\mathcal{X}, \mathcal{D}, \mathcal{F})$ avec $\mathcal{X} = \{X_1, \ldots, X_n\}$, un ensemble de variables, $\mathcal{D} = \{D_{X_1}, \ldots, D_{X_n}\}$, un ensemble de domaines finis de taille maximale $d = \max_{i=1}^{n} d_{X_i}$ ($d_{X_i} = |D_{X_i}|$), et $\mathcal{F} = \{f_1, \ldots, f_e\}$, un ensemble de fonctions à valeurs réelles positives, chacune est définie sur un sous-ensemble de variables $\mathbf{S}_i \subseteq \mathcal{X}$ (i.e., la portée). La distribution jointe est définie par :*

$$\mathbb{P}(\mathcal{X}) = \frac{\prod_{i=1}^{e} f_i(\mathbf{S}_i)}{\sum_{\mathcal{X}} \prod_{i=1}^{e} f_i(\mathbf{S}_i)}$$

Les réseaux Bayésiens représentent un cas particulier utilisant une probabilité conditionnelle par variable ($e = n$) et la constante de normalisation (dénominateur de $\mathbb{P}(\mathcal{X})$) vaut 1. Le problème *Most Probable Explanation* (MPE) consiste à trouver l'affectation la plus probable sur toutes les variables de $\mathcal{X}$ maximisant $\mathbb{P}(\mathcal{X})$.

Un *Problème de Satisfaction de Contraintes* (CSP) est un triplet $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ avec $\mathcal{X}$,$\mathcal{D}$ définis comme pour les PGM, et $\mathcal{C}$, un ensemble de contraintes. Chaque contrainte est définie comme une fonction booléenne sur un sous ensemble de variables $\mathbf{S}_i \subseteq \mathcal{X}$ indiquant les tuples dans $D_{\mathbf{S}_i} = \prod_{X \in \mathbf{S}_i} D_X$ sont autorisées ou non. Le problème consiste à trouver une *affectation possible* de $\mathcal{X}$, satisfaisant toutes les contraintes. Un *Problème de Satisfaction de Contraintes Pondérées* (WCSP) est une extension des CSP pour l'optimisation.

Définition 2 (WCSP)**.** *Un WCSP est un triplet $(\mathcal{X}, \mathcal{D}, \mathcal{W})$ avec $\mathcal{X}, \mathcal{D}$ définis comme pour les PGM et $\mathcal{W} = \{w_1, \ldots, w_e\}$, un ensemble de* fonctions de coûts*. Chaque fonction de coûts est définie sur un sous-ensemble de variables et prend des valeurs entières positives dans $E^+ = \mathbb{N} \cup \{\top\}$ ($\top = +\infty$ pouvant être associé aux affectations interdites[2]). Le but est de trouver une affectation valide de $\mathcal{X}$ minimisant $\sum_{i=1}^{e} w_i(\mathbf{S}_i)$.*

Un PGM $(\mathcal{X}, \mathcal{D}, \mathcal{F})$ peut être traduit en un WCSP $(\mathcal{X}, \mathcal{D}, \mathcal{W})$ avec $\forall i \in [1, e], w_i(\mathbf{S}_i) = \lceil -M \log(f_i(\mathbf{S}_i)) + C \rceil$ (M, C deux constantes pour la précision et la positivité des w_i lors de la transformation des réels en entiers) , il préserve l'ensemble des solutions optimales si une valeur suffisamment grande est utilisée pour M.

Dans la suite du papier, nous utilisons f pour représenter les fonctions de coûts. Etant donnée une affectation $t \in D_{\mathcal{X}}$, $t[\mathbf{S}]$ représente l'affectation t sur les variables de $\mathbf{S}$. Soit $f = f_1 + f_2$ la *somme (jointure) de deux fonctions de coûts* définie par $f(t) = f_1(t[\mathbf{S}_1]) + f_2(t[\mathbf{S}_2]), \forall t \in D_{\mathbf{S}_1 \cup \mathbf{S}_2}$. Soit $f = f_1 - f_2$ la *soustraction de deux fonctions de coûts* telle que $\mathbf{S}_2 \subseteq \mathbf{S}_1$ et $f(t) = f_1(t) - f_2(t[\mathbf{S}_2]), \forall t \in D_{\mathbf{S}_1}$ avec $\top - \top = \top$. $f[\mathbf{S}']$ représente *la projection d'une fonction de coûts* sur le sous-ensemble $\mathbf{S}'$ des variables de $\mathbf{S}$ telle que $\mathbf{S}' \subseteq \mathbf{S}$ et $\forall t' \in D_{\mathbf{S}'} f[\mathbf{S}'](t') = \min_{t \in D_{\mathbf{S}} \text{ s.t. } t[\mathbf{S}'] = t'} f(t)$. Une fonction de coûts sur deux (resp. une) variables est appelée une fonction de coûts *binaire* (resp. *unaire*). Une *fonction de coûts vide* a tous ses coûts égaux à zéro et est supprimée du WCSP.

L'algorithme de séparation évaluation (Depth-First Branch and Bound DFBB) est une méthode complète de recherche arborescente pour résoudre les WCSP. Les méthodes de cohérences locales souples produisent de fortes bornes inférieures pour DFBB par application de *Transformations Préservant l'Equivalence* (Equivalence Preserving Transformation EPT) [1]. Une *arc EPT* ajoute la projection $f[X]$ d'une fonction de coûts à une fonction de coûts unaire $f_1(X), X \in \mathbf{S}$ et soustrait $f[X]$ à f (*i.e.* remplace f par $f - f[X]$[3]) afin de garder un problème équivalent. L'application de la *cohérence d'arc souple* consiste à utiliser les arc

[1] Un exemple connu *le réseau de ségrégation* [3], où $\mathbb{P}(G_{i,jp} \mid G_{a,jp}, G_{a,jm}, S_{i,jp}) = 2\mathbb{P}(G_{i,jp} \mid G_{a,jp}, S_{i,jp})\mathbb{P}(G_{i,jp} \mid G_{a,jm}, S_{i,jp})$ car $(G_{a,jp} \perp\!\!\!\perp G_{a,jm} \mid G_{i,jp}, S_{i,jp})$, mais on ne peut pas le représenter comment un réseau bayésien.

[2] Par simplicité, nous utilisons $\top = +\infty$, mais nos résultats restent valides pour un $\top$ fini.

[3] $f - f[X]$ est bien une fonction de coûts toujours positive

EPT pour toutes les fonctions de coûts et toutes les directions ($\forall f(\mathbf{S}) \in \mathcal{W}, \forall X \in \mathbf{S}$) jusqu'à ce que toutes les projections soient vides. Une *arc EPT directionnelle* ajoute une projection $(f + f_2)[X]$ d'une fonction de coût binaire $f(X,Y)$ et unaire $f_2(Y)$ vers $f_1(X)$ et soustrait le résultat de la projection. L'application de la *cohérence d'arc souple directionnelle* (DAC) consiste à utiliser les arc EPT directionnelles sur toutes les fonctions de coûts suivant une seule direction ($X < Y$), indiquée par un ordre total sur $\mathcal{X}$, ainsi la terminaison est assurée. DFBB peut être également amélioré en utilisant l'*élimination de variables* à la volée [7]. Soient X une variable, $F = \{f_i \in \mathcal{W} \text{ s.t. } X \in \mathbf{S}_i\}$ et $\mathbf{S} = \bigcup_{f_i \in F} \mathbf{S}_i$. L'élimination de la variable X consiste à remplacer X et F par la projection $(\sum_{f \in F} f)[\mathbf{S} \setminus \{X\}]$ dans le WCSP courant. L'élimination de variables est normalement exponentielle en temps et en espace par rapport à la taille de $\mathbf{S}$ et est peu coûteuse si $|\mathbf{S}|$ est petite ou si X est affectée ou s'il est connecté au reste du problème par une seule fonction de coûts ou à travers au moins une contrainte bijective (contrainte d'égalité par exemple). DFBB appliquant l'élimination de variables i-bornée (toutes les variables avec $|\mathbf{S}| \leqslant i$ sont éliminées) et les cohérences locales souples (EDAC[4] pour les fonction de coûts binaires et ternaires [13]) est noté DFBB-VE(i) dans les résultats expérimentaux.

3 Décomposition de paire d'une fonction de coûts

Nous définissons tout d'abord la notion de *décomposition par paire*.

Définition 3. *Une* décomposition par paire *d'une fonction de coûts $f(\mathbf{S})$ par rapport à deux variables $X, Y \in \mathbf{S}$ ($X \neq Y$), est une réecriture de f en une somme de deux fonctions (positives) $f_1(\mathbf{S} \setminus \{Y\})$ et $f_2(\mathbf{S} \setminus \{X\})$ telles que :*

$$f(\mathbf{S}) = f_1(\mathbf{S} \setminus \{Y\}) + f_2(\mathbf{S} \setminus \{X\})$$

Un exemple de fonction décomposable est donnée par la Figure 1. Une décomposition par paire remplace une fonction de coûts d'artié $r = |\mathbf{S}|$ avec des fonctions de coûts d'arité plus petites $(r-1)$. Ce processus de décomposition peut être répété récursivement sur les fonctions résultantes, jusqu'à ce qu'aucune décomposition par paire soit trouvée pour chaque fonction de coûts restante. Une fonction de coûts $f(\mathbf{S})$ qui ne peut être décomposée par paire pour aucun choix de $X, Y \in \mathbf{S}$ est dite *non décomposable*. Un WCSP est *décomposé par paire* si toutes ses fonctions de coûts sont non décomposables. Cette propriété est respectée en appliquant le processus itératif précédemment décrit pour toutes les fonctions de coûts.

[4] EDAC combine AC et DAC. De plus elle teste pour chaque variable DAC avec un ordre mettant cette variable en premier sur le sous-problème constitué de $F = \{f_i \in W_{\mathbf{S}} \text{ t.q. } x \in \mathbf{S}_i$

W	X	Y	U	f
1	1	1	1	5
1	1	1	2	$\top$
1	1	2	1	$\top$
1	1	2	2	3
1	2	1	1	6
1	2	1	2	5
1	2	2	1	$\top$
1	2	2	2	$\top$
2	1	1	1	4
2	1	1	2	$\top$
2	1	2	1	$\top$
2	1	2	2	2
2	2	1	1	2
2	2	1	2	1
2	2	2	1	3
2	2	2	2	1

$=$

W	X	Y	f_1
1	1	1	5
1	1	2	3
1	2	1	5
1	2	2	$\top$
2	1	1	4
2	1	2	2
2	2	1	1
2	2	2	1

$+$

X	Y	U	f_2
1	1	1	0
1	1	2	$\top$
1	2	1	$\top$
1	2	2	0
2	1	1	1
2	1	2	0
2	2	1	2
2	2	2	0

FIG. 1 – $f(W,X,Y,U)$ fonction sur quatre variables Booléennes est décomposable par rapport à W,U comme $f_1(W,X,Y) + f_2(X,Y,U)$.

Le théorème suivant montre que tester si une fonction de coûts peut être décomposée par paire par rapport à X,Y est équivalent à tester l'indépendance de paire entre X et Y dans une distribution de probabilité appropriée.

Théorème 1 (Equivalence entre l'indépendance de paire et décomposition)**.** *Soit $\mathbf{S} = \{X,Y\} \cup \mathbf{Z}$ un ensemble de variables aléatoires, $f(\mathbf{S})$ une fonction de coûts sur $\mathbf{S}$ avec au moins une affectation autorisée, et une distribution $\mathbb{P}_f = \frac{1}{\sum_{\mathbf{S}} \exp(-f(\mathbf{S}))} \exp(-f(\mathbf{S}))$ (une* distribution de Gibbs *paramétrisée par f). $(X \perp\!\!\!\perp Y \mid \mathbf{Z})$ représente que X et Y sont indépendants par paire étant données toutes les autres variables de $\mathbb{P}_f$. Alors,*

$$(X \perp\!\!\!\perp Y \mid \mathbf{Z}) \iff f(X,\mathbf{Z},Y) = f_1(X,\mathbf{Z}) + f_2(\mathbf{Z},Y)$$

Au lieu de tester l'indépendance de paire dans $\mathbb{P}_f$, qui nécessite des sommations et des multiplications des nombres réels, nous proposons un test plus simple pour la décomposabilité de paire basée sur des égalités de différences de coûts. Pour une fonction de coûts décomposable par paire $f(X, \mathbf{Z}, Y) = f_1(X, \mathbf{Z}) + f_2(\mathbf{Z}, Y)$ n'ayant que des coûts finis et pour tout tuple $z \in D_{\mathbf{Z}}$, si nous considérons toutes les paires de valeurs k, l pour Y alors la différence $f(x, z, k) - f(x, z, l) = (f_1(x, z) + f_2(z, k)) - (f_1(x, z) + f_2(z, l)) = (f_2(z, k) - f_2(z, l))$ ne dépend pas de x. Lorsque $f(X, \mathbf{Z}, Y)$ n'est pas limité à des coûts finis, la soustraction d'un coût infini est mal définie. Par exemple, si $f_2(z, k)$ et $f_2(z, l)$ sont tous les deux égaux à $\top$ alors $f(x, z, k) = f(x, z, l) = \top$ pour tout x. Donc $f_1(x, z)$ peut prendre *n'importe quelle* valeur et la décomposition est toujours valable. Les coûts infinis offrent une liberté supplémentaire dans la décomposition et doivent être traités spécifiquement.

Pour avoir un test qui peut exploiter les coûts infinis, nous introduisons une extension de la structure de valuation des coûts $E = \mathbb{Z} \cup \{-\top, \top, \Omega\}$ incluant les coûts négatifs et un élément spécial asborbant Ω qui capture la liberté générée par la soustraction des coûts infinis. Soit $a \boxminus b$ représentant la différence de deux éléments $a, b \in E : a \boxminus b = a - b$, sauf pour $\top \boxminus \top = -\top \boxminus -\top = a \boxminus \Omega = \Omega \boxminus b = \Omega$, $\top \boxminus -\top = \top \boxminus c = c \boxminus -\top = \top$, $-\top \boxminus \top = -\top \boxminus c = c \boxminus \top = -\top$ avec $c \in \mathbb{Z}$. La comparaison de deux éléments $a, b \in E$ noté par $a \triangleq b$ est vraie si et seulement si (1) $a, b \in \mathbb{Z} \cup \{\top, -\top\}$ et $a = b$, ou (2) $a = \Omega$ ou $b = \Omega$. Nous avons alors :

Théorème 2. *Une fonction de coûts $f(X, \mathbf{Z}, Y)$ est décomposable par paire par rapport à X, Y ssi $\forall z \in D_{\mathbf{Z}}, \forall k, l \in D_Y (k < l)$[5], $\forall u, v \in D_{\mathbf{X}}$:*

$$f(u, z, k) \boxminus f(u, z, l) \triangleq f(v, z, k) \boxminus f(v, z, l)$$

Remarque 1. *Puisque $f(u, z, k) \boxminus f(u, z, l) = \Omega \triangleq a$, $\forall a \in E$, il suffit de trouver une valeur $u \in D_{\mathbf{X}}$ telle que $f(u, z, k) \boxminus f(u, z, l) \neq \Omega$ et de tester $\forall v \in D_{\mathbf{X}}$ $(v \neq u)$ telle que $f(v, z, k) \boxminus f(v, z, l) \neq \Omega$*
$f(u, z, k) \boxminus f(u, z, l) = f(v, z, k) \boxminus f(v, z, l)$
Puisque l'égalité est transitive le théorème 2 est bien vérifié.

Exemple 1. *Dans la Figure 1, nous avons $f(W, X, Y, U)$ décomposable par paire par rapport à W et U car :*

$$\begin{array}{ccccccc} f(1111) & \boxminus & f(1112) & \triangleq & f(2111) & \boxminus & f(2112) \\ 5 & \boxminus & \top & \triangleq & 4 & \boxminus & \top \\ f(1121) & \boxminus & f(1122) & \triangleq & f(2121) & \boxminus & f(2122) \\ \top & \boxminus & 3 & \triangleq & \top & \boxminus & 2 \\ f(1211) & \boxminus & f(1212) & \triangleq & f(2211) & \boxminus & f(2212) \\ 6 & \boxminus & 5 & \triangleq & 2 & \boxminus & 1 \\ f(1221) & \boxminus & f(1222) & \triangleq & f(2221) & \boxminus & f(2222) \\ \top & \boxminus & \top & \triangleq & 3 & \boxminus & 1 \end{array}$$

Ensuite, nous montrons comment contruire f_1, f_2 d'une fonction de coûts décomposable par paire en utilisant les projections et les soustractions des fonctions de coûts (voir la Figure 1 pour cette application).

Théorème 3. *Soit $f(X, \mathbf{Z}, Y)$ décomposable par paire par rapport à X, Y. Alors $f_1(X, \mathbf{Z}) = f[X, \mathbf{Z}]$ et $f_2(\mathbf{Z}, Y) = (f - f_1)[\mathbf{Z}, Y]$ est une décomposition valide de f, i.e. $f = f_1 + f_2$.*

Pour appliquer la décomposition par paire sur une fonction de coûts $f(\mathbf{S})$ avec $r = |\mathbf{S}|$ peut nécessiter $\frac{r(r-1)}{2}$ tests, *i.e.* vérifier pour toutes les paires de variables de $\mathbf{S}$. Et il peut produire des fonctions (non vides) d'arité (r-1). La répétition de la décomposition par paire pour les fonctions de coûts résultantes produira au plus $\min(\binom{r-p}{r}, 2^p)$ fonctions de coûts d'arité $(r - p)$ avec des portées différentes. Chaque test est linéaire par rapport à la taille des fonctions de coûts $(r - p)$-aire en $O(d^{(r-p)+1})$. Donc la complexité globale dans le pire des cas de l'application de la décomposition de paire pour e fonctions de coûts originales avec une arité r maximale est $O(e \sum_{p=0}^{r-1} 2^p (r-p)^2 d^{(r-p)+1}) = O(e \sum_{p=0}^{r-1} (r-p)^2 d^{r+1}) = O(e \frac{r(r+1)(2r+1)}{6} d^{r+1}) = O(er^3 d^{r+1})$ en temps et $O(e \max_{p=0}^{r-1} 2^p d^{(r-p)}) = O(ed^r)$ en espace (soit linéaire par rapport à la taille du problème).

A chaque étape, le choix des variables utilisées pour la décomposition et la manière de répartir les coûts dans f_1 et f_2 peuvent influencer la décomposabilité de f_1 et f_2 à l'itération suivante. Sous certaines conditions, il est possible de trouver une séquence de décomposition de paire (la paire de variables à sélectionner dans f et comment distribuer f dans f_1 et f_2) qui mène à une décomposition optimale (avec une arité minimale).

Propriété 1 (Décomposition minimale unique pour des fonctions de coûts finis [4])**.** *Une fonction de coûts avec seulement des coûts finis admet une décomposition* unique et minimale.

Ebauche de preuve. Si une fonction de coûts $f(\mathbf{S})$ a uniquement des coûts finis, alors sa distribution de probabilité associée $\mathbb{P}_f$ (voir Théorème 1) est positive. Le théorème d'*Hammersley & Clifford* [4] [6] dit qu'une

[5] Chaque élément de E admet un oppposé donc $-(f(u, z, k) \boxminus f(u, z, l)) = f(u, z, l) \boxminus f(u, z, k)$ d'où la condition de retester que dans un sens les égalités en prenant un ordre arbitraire sur les valeurs

[6] Voir aussi le Théorème 4.5 page 121 dans [6].

distribution positive se factorise en un unique réseau de Markov minimal $\mathcal{H}$. Le réseau est minimal dans le sens où la suppression d'une arête crée une nouvelle indépendance conditionnelle non présente dans $\mathbb{P}_f$. En prenant les cliques maximales dans $\mathcal{H}$, chaque clique donne la portée d'un facteur (une fonction positive) et $\mathbb{P}_f$ est égale au produit de ces facteurs divisé par une constante de normalisation. Cette factorisation est équivalente à une somme de fonctions de coûts en prenant $-\log(\mathbb{P}_f)$ (voir la preuve du Théorème 1). De cette décomposition résultante, il est facile de construire une séquence inverse de fonctions de coûts (positives) valides jusqu'à atteindre la fonction de coûts f originale. □

Cependant, dès qu'il y a des coûts infinis, le théorème précédent ne peut pas s'appliquer. Dans ce cas, nous proposons une approche heuristique dont le but est d'accumuler les coûts sur les premières variables dans l'ordre des variables de DAC. Cela devrait améliorer des bornes inférieures basées sur DAC. Nous testons donc des paires de variables dans $f(\mathbf{S})$ dans l'ordre DAC inverse[7] et projetons les fonctions de coûts (Théorème 3) sur la portée contenant les premières variables dans l'ordre DAC en premier. Dans la Figure 1, supposons l'ordre (W,X,Y,U), cela correspond à projeter en premier sur $\{W, X, Y\}$. Nous montrons que notre approche heuristique est capable de trouver une décomposition optimale pour quelques fonctions particulières.

Propriété 2 (Identification de structure d'arbre pour les contraitnes dures [11])**.** *Une fonction de coûts avec seulement des coûts infinis qui admet une décomposition en arbre de contraintes binaires est identifiable par notre stratégie de décomposition.*

Ce résultat vient de [11] et le fait qu'une décomposition par paire par rapport à X, Y supprimera une arête redondante $\{X, Y\}$ dans le réseau minimal[8] sans affecter l'ensemble des solutions. Par exemple, une contrainte globale d'égalité sur $\mathbf{S}$ sera décomposée en un arbre de contraintes d'égalité binaires : $f(\mathbf{S}) = \sum_{Y \in \mathbf{S} \setminus \{X\}} f_Y(X, Y)$ avec $f_Y(X, Y) \equiv (X = Y)$.

De la même façon, une fonction de coûts linéaire $f(X_1, \ldots, X_r) = \sum_{i=1}^{r} a_i X_i$ peut être décomposée en une somme de r fonctions de coûts unaires[9].

[7] Nous testons Y, U avant W, X si $\max(Y, U) > \max(W, X) \vee (\max(Y, U) = \max(W, X) \wedge \min(Y, U) > \min(W, X))$.

[8] Dans le cadre des CSP, le CSP binaire, appelé le *réseau minimal*, qui est la meilleure approximation d'une relation non binaire, est définie par les projections de la relation sur toutes les paires de variables [11].

[9] Notons que l'arc cohérence souple ferait de même si la fonction de coûts vide résultante est supprimée après les projections.

Travaux existants

Les travaux existants qui ont considéré la factorisation des fonctions de probabilités spécifiques comme les *noisy-or* et ses généralisations [5], ou des factorisations générales dédiées à l'inférence probabiliste [15]. Comparés à notre approche, ces travaux ajoutent de nouvelles variables. Une autre approche possible pour diminuer l'arité des fonctions de coûts est de passer à leur représentation duale [12]. Cela a été tester dans [2] pour Max-2SAT et était apparemment inefficace. Cela devrait probablement empirer pour de plus grands domaines.

4 Projection&Soustraction sur les fonctions de coûts binaires

Quand un WCSP est décomposé par paire, il est possible d'inférer une information supplémentaireà partir de la fonction de coûts résultante non binaire par projection & soustraction sur des fonctions de coûts de plus petites arités. Nous choisissons de projeter chaque fonction de coûts non binaire sur toutes les fonctions de coûts binaires possibles étant donnée la portée de la fonction en suivant un ordre des projections&soustractions compatible avec l'ordre des variables DAC (*i.e.* on projette d'abord sur les paires de variables avec les plus petites positions dans l'ordre DAC). Chaque projection $f_i(X, Y) = f[X, Y]$ est suivie par une soustraction $f = f - f_i$ afin de préserver l'équivalence du problème. Notons que f peut être vide après ces soustractions et est alors supprimée du problème. Notons aussi que la même fonction binaire peut recevoir des projections de coûts de plusieurs fonctions non binaires ayant des portées se chevauchant, résultant d'une plus forte inférence. La complexité dans le pire des cas de *projeter&soustraire* appliquée sur e fonctions de coûts avec r comme arité maximale est $O(er^2d^r)$ en temps et $O(er^2d^2)$ en espace.

Exemple 2. *Dans l'exemple de la Figure 1,* $f_1(W, X, Y) = b_1(W, X) + b_2(X, Y) + f_3(W, X, Y)$ *et* $f_2(X, Y, U) = b_3(X, U) + b_4(Y, U) + f_4(X, Y, U)$. *Finalement, une borne inférieure égale à 1 est déduite par arc cohérence souple appliquée sur* b_1.

W	X	b_1
1	1	3
1	2	5
2	1	2
2	2	1

X	Y	b_2
1	1	2
1	2	0
2	1	0
2	2	0

	i=2	i=3	i=4	i=5	i=6	i=7
Linkage (22)						
DFBB-VE(i)	14	14	15	13	12	10
DFBB-VE(i)+dec	16	**19**	17	13	14	13
DFBB-VE(i)+ps	**17**	16	17	18	16	16
DFBB-VE(i)+dec+ps	16	18	**19**	**19**	**19**	**17**
Grids (32)						
DFBB-VE(i)	28	28	25	24	17	18
DFBB-VE(i)+dec	29	29	29	28	28	31
DFBB-VE(i)+ps	28	28	28	23	25	24
DFBB-VE(i)+dec+ps	**31**	**30**	**31**	**31**	**32**	**32**

TAB. 1 – Nombre d'instances résolues en moins d'une heure par chaque méthode

X	U	b_3
1	1	0
1	2	0
2	1	1
2	2	0

Y	U	b_4
1	1	0
1	2	0
2	1	1
2	2	0

W	X	Y	f_3
1	1	1	0
1	1	2	0
1	2	1	0
1	2	2	$\top$
2	1	1	0
2	1	2	0
2	2	1	0
2	2	2	0

X	Y	U	f_4
1	1	1	0
1	1	2	$\top$
1	2	1	$\top$
1	2	2	0
2	1	1	0
2	1	2	0
2	2	1	0
2	2	2	0

5 Résultats expérimentaux

DFBB-VE(i)[10] maintenant EDAC et utilisant un ordre dynamique d'affectation des variables basé sur les conflits et une pondération des fonctions de coûts [9] a obtenu les meilleurs résultats de l'évaluation de MPE de UAI'08[11] (et UAI'10[12] pour le cas à 20 minutes) excepté pour deux benchmarks difficiles : les réseaux d'analyse de liaison et les réseaux en grille.

Les réseaux d'analyse de liaison génétique (Linkage). Les instances ont entre 334 et 1289 variables. Le domaine maximal de taille d est entre 3 et 7 et l'arité la plus grande est 5. Cette famille de benchmarks est constituée de 22 problèmes.

Les réseaux en grille (Grids). Chaque problème est une grille $l \times l$ et chaque ternaire CPT est générée aléatoirement et uniformément. 50, 75 or 90% des CPT sont déterministes et les variables sont Booléennes. Pour ces instances l varie de 12 à 50 (*i.e.* $n = 2500$ variables). Cette famille de benchmarks a 32 problèmes.

Les expérimentations[10] ont été réalisées sur un ordinateur 2.6 GHz Intel Xeon avec 4GB sous Linux 2.6. Les temps CPU totaux sont en secondes et limités à 1 heure pour chaque instance ("-" représente un dépassement de temps). Aucune borne supérieure n'est donnée au départ. Nous testons l'application de la décomposition par paire (DFBB-VE(i)+dec), projeter&soustraire les fonctions de coûts n-aires(DFBB-VE(i)+ps), et la combinaison des deux techniques (DFBB-VE(i)+dec+ps). La décomposition par paire, de projeter&soustraire et l'élimination de variables de degré $|\mathbf{S}| \leq i$ sont effectuées seulement en prétraitement. L'élimination à la volée des variables de degré au plus 2 est réalisé durant la recherche. Toutes les méthodes utilisent l'heuristique *MinFill* pour l'ordre d'élimination de variables et l'ordre inverse est utilisé pour DAC. La Table 1 résume le nombre de problèmes résolus par les différentes versions de DFBB-VE(i). La décomposition par paire et projeter&soustraire résolvent plus de problèmes que DFBB-VE(i) seul et leur combinaison obtient les meilleurs résultats, sauf pour les Linkage avec $i = 2, 3$. Remarquablement, toutes les instances de grilles sont résolues avec de l'élimination de variables bornée pour un i suffisamment grand. montrant l'effet de la décomposition par paire, spécialement quand il y a suffisament de déterminisme. Une factorisation similaire est observée dans [14] pour les CSP.

Nous avons ensuite comparé la meilleure méthode avec AND/OR Branch and Bound avec cache et minibucket statique j-borné (AOBB-C+SMB(j)) qui a eu les meilleurs résultats pour Linkage et Grids à l'évaluation UAI'08. Cette méthode est implémentée dans `aolibWCSP`[13]. La valeur de j est choisie comme dans [10]. La Table 2 donne les tailles des problèmes (n, d), la treewidth (w) et résume le temps CPU total utilisé pour les différentes méthodes avec le i correspondant (en choisissant $i \in [2, 7]$ permettant d'obtenir les

[10] `mulcyber.toulouse.inra.fr/projects/toulbar2` version 0.9.4.
[11] `graphmod.ics.uci.edu/uai08/Evaluation`
[12] `www.cs.huji.ac.il/project/UAI10`
[13] `graphmod.ics.uci.edu/group/aolibWCSP`

Problème				DFBB-VE(i)			DFBB-VE(i) +dec+ps			AOBB-C+SMB(j) +VE(i)				AOBB-C+SMB(j) +VE(i)+dec+ps	
	n	d	w	time (s)	i	n'	time (s)	i	n'	j	i	time (s)	n'	time (s)	n'
Linkage															
ped1	334	4	16	0.12	3	159	**0.07**	3	142	10	3	0.1	159	0.08	142
ped7	1068	4	38	4.04	2	375	**1.18**	4	213	20	4	1915.56	274	131.14	213
ped9	1118	4	31	-			**3.36**	6	147	20	6	-	179	104.62	147
ped18	1184	5	24	149.71	4	365	**3.19**	5	213	20	5	54.67	255	18.82	213
ped19	793	5	32	-			-			20	6	-	337	-	304
ped20	437	5	23	3.46	4	95	**0.39**	6	42	16	6	407.6	68	66.53	42
ped23	402	5	26	0.09	4	79	**0.05**	3	98	12	3	6.05	114	1.52	98
ped25	1289	5	29	1207.97	4	269	**0.65**	6	102	20	6	25.91	158	32.51	102
ped30	1289	5	24	543.20	2	633	**5.44**	5	213	20	5	28.39	272	10.10	231
ped33	798	4	30	0.84	3	272	**0.54**	6	151	18	6	21.33	175	8.35	151
ped34	1160	5	36	1.13	2	326	**0.36**	5	167	20	5	-	196	24.56	167
ped37	1032	4	22	0.21	5	103	**0.11**	4	120	10	4	87.18	141	9.58	120
ped38	724	5	18	0.24	5	122	**0.18**	5	97	12	5	137.60	122	124.70	97
ped39	1272	5	22	16.68	4	183	**0.24**	5	107	18	5	6.87	148	2.60	107
ped41	1062	5	35	-			**302.05**	4	308	20	4	-	372	1271.31	308
ped42	448	5	24	1.94	4	140	**0.34**	6	81	16	6	240.94	95	155.39	81
ped44	811	4	31	-			505.46	5	215	20	5	2631.71	248	**333.89**	215
ped50	514	6	18	0.90	4	133	**0.18**	4	118	12	4	316.92	133	521.92	118
Grids															
90-24-1	576	2	35	0.07	3	566	**0.04**	3	291	18	3	2323.01	566	0.63	291
90-26-1	676	2	41	0.29	3	668	**0.07**	3	500	16	3	2821.66	668	20.16	500
90-30-1	900	2	49	5.50	2	766	**0.26**	4	382	18	4	-	766	3.57	382
90-34-1	1156	2	63	328.42	2	1052	**0.48**	5	518	20	5	-	1037	14.41	518
90-38-1	1444	2	64	-			1.96	6	249	20	6	-	983	**0.64**	249

TAB. 2 – Comparaison avec/sans le pré-traitement dec+ps pour DFBB-VE(i) et AOBB-C+SMB(j)

meilleurs résultats pour chaque instance) et j. Le pré-traitement "dec+ps" améliore globalement les résultats de DFBB-VE(i) et AOBB-C+SMB(j). Le temps du pré-traitement était toujours inférieur à une seconde dans nos expérimentations (n', nombre de variables après le pré-traitement). DFBB-VE(i)+dec+ps a obtenu les meilleurs résultats pour tous les problèmes sauf pour ped44 and 90-38-1. Notons que la variation de i pour DFBB-VE(i)+dec+ps est plus petite que celle de j pour AOBB-C+SMB(j)[14].

6 Conclusion

La décomposition par paire combinée avec les projections sur les fonctions de coûts binaires et l'élimination de variables est une puissante technique à l'intérieur de DFBB pour MPE. Les travaux futurs devraient être fait sur la décomposition par paire approchée. D'autres expérimentations sur d'autres problèmes incluant l'inférence probabiliste devraient également être réalisées.

[14]Un bon moyen de régler la valeur de i est de prendre la valeur correspondant au maximum de la distribution des degrés pour le modèle graphique considéré (*i.e.* $i = 5$ pour Linkage et $i = 6$ pour Grids).

A Preuve du Théorème 1

Dans la suite, nous utilisons $\mathbb{P}$ comme raccourci pour $\mathbb{P}_f$ et $C = \sum_{\mathbf{S}} \exp(-f(\mathbf{S})) > 0$, une constante de normalisation. Par définition des indépendances conditionnelles [8], nous avons :
$(X \perp\!\!\!\perp Y \mid \mathbf{Z}) \iff \mathbb{P}(X, Y, \mathbf{Z}) = \mathbb{P}(X \mid \mathbf{Z})\mathbb{P}(Y \mid \mathbf{Z})\mathbb{P}(\mathbf{Z})$
$\iff \mathbb{P}(X, Y, \mathbf{Z}) = \frac{\mathbb{P}(X,\mathbf{Z})}{\mathbb{P}(\mathbf{Z})}\mathbb{P}(Y, \mathbf{Z})$, with $\mathbb{P}(\mathbf{Z}) > 0$.

$\boxed{\Longrightarrow}$
Supposons $\mathbb{P}(X, Y, \mathbf{Z}) = \mathbb{P}(X \mid \mathbf{Z})\ \mathbb{P}(Y \mid \mathbf{Z})\ \mathbb{P}(\mathbf{Z})$. Nous avons $f(X, \mathbf{Z}, Y) = -\log(C\ \mathbb{P}(X, Y, \mathbf{Z})) = -\log(C\ \mathbb{P}(X \mid \mathbf{Z})\ \mathbb{P}(Y \mid \mathbf{Z})\ \mathbb{P}(\mathbf{Z}))$. Définissons $f'_1(X, \mathbf{Z}) = -\log(\mathbb{P}(X \mid \mathbf{Z}))$ et $f'_2(\mathbf{Z}, Y) = -\log(\mathbb{P}(Y \mid \mathbf{Z})\mathbb{P}(\mathbf{Z}))$, deux fonctions de coût positives.
Ainsi $f(X, \mathbf{Z}, Y) = f'_1(X, \mathbf{Z}) + f'_2(\mathbf{Z}, Y) - \log(C)$. Car f est positive, nous avons $\forall x \in D_X, \forall y \in D_Y, \forall z \in D_{\mathbf{Z}}, f'_1(x, z) + f'_2(z, y) - \log(C) \geq 0$. Si $\log(C)$ est négatif, nous ajoutons $-\log(C)$ à f'_1 ou f'_2 afin d'obtenir f_1, f_2.
Sinon, pour chaque $z \in D_{\mathbf{Z}}$, nous décomposons $\log(C) = c_z^1 + c_z^2$ en deux nombres positifs.
Soit $\hat{x}_z = \text{argmin}_{x \in D_X} f'_1(x, z)$ et $\hat{y}_z = \text{argmin}_{y \in D_Y} f'_2(z, y)$. De plus, nous avons $f'_1(\hat{x}_z, z) + f'_2(z, \hat{y}_z) - \log(C) = f(\hat{x}_z, z, \hat{y}_z) \geq 0$. Une solution possible est $c_z^1 = \min(f'_1(\hat{x}_z, z), \log(C))$ et $c_z^2 = \log(C) - c_z^1$.

Either $c_z^1 = \log(C) \leq f_1'(\hat{x}_z, z)$ et $c_z^2 = 0$, ou $c_z^1 = f_1'(\hat{x}_z, z)$ et $c_z^2 = \log(C) - f_1'(\hat{x}_z, z)$.
Dans ce cas, $f_2'(z, \hat{y}_z) - c_z^2 = f_2'(z, \hat{y}_z) + f_1'(\hat{x}_z, z) - \log(C) \geq 0$, ainsi $\forall y \in D_Y, f_2'(z, y) - c_z^2 \geq 0$. Nous définissons $\forall x, y, z,\ f_1(x, z) = f_1'(x, z) - c_z^1$ et $f_2(z, y) = f_2'(z, y) - c_z^2$, qui sont des nombres positifs Finalement, nous en déduisons $f(X, \mathbf{Z}, Y) = f_1(X, \mathbf{Z}) + f_2(\mathbf{Z}, Y)$.

$\boxed{\Longleftarrow}$
Supposons que nous ayons trois fonctions de coûts $f(X, \mathbf{Z}, Y), f_1(X, \mathbf{Z}), f_2(\mathbf{Z}, Y)$ telles que $f(X, \mathbf{Z}, Y) = f_1(X, \mathbf{Z}) + f_2(\mathbf{Z}, Y)$.
Nous avons
$\exp(-f(X, \mathbf{Z}, Y)) = \exp(-f_1(X, \mathbf{Z})) \exp(-f_2(\mathbf{Z}, Y))$.
Par marginalisation, nous avons
$\mathbb{P}(X, \mathbf{Z}) = \sum_Y \mathbb{P}(X, Y, \mathbf{Z}) = \frac{1}{C} \sum_Y \exp(-f(X, \mathbf{Z}, Y))$
$= \frac{1}{C} \exp(-f_1(X, \mathbf{Z})(\sum_Y \exp(-f_2(\mathbf{Z}, Y)))$, $\mathbb{P}(Y, \mathbf{Z})$
$= \sum_X \mathbb{P}(X, Y, \mathbf{Z})$
$= \frac{1}{C} \exp(-f_2(\mathbf{Z}, Y))\ (\sum_X \exp(-f_1(X, \mathbf{Z})))$
et $\mathbb{P}(\mathbf{Z}) = \sum_{X,Y} \mathbb{P}(X, Y, \mathbf{Z})$
$= \frac{1}{C}(\sum_X \exp(-f_1(X, \mathbf{Z})))(\sum_Y \exp(-f_2(\mathbf{Z}, Y)))$.
Finalement, nous en déduisons $\frac{\mathbb{P}(X, \mathbf{Z})}{\mathbb{P}(\mathbf{Z})} \mathbb{P}(Y, \mathbf{Z}) = \frac{1}{C} \exp(-f_1(X, \mathbf{Z})) \exp(-f_2(\mathbf{Z}, Y)) = \mathbb{P}(X, Y, \mathbf{Z})$.

B Preuve du Théorème 2

Il est toujours possible de décomposer une fonctiond de coûts f en trois fonction de coûts (positives) $f(X, \mathbf{Z}, Y) = f_1(X, \mathbf{Z}) + f_2(\mathbf{Z}, Y) + \Delta(X, \mathbf{Z}, Y)$ (f_1, f_2 peuvent être égales à la fonction constante nulle). Une fonction de coût non nulle est appelée *réductible* si f_1 ou f_2 sont des fonctions de coûts non nulles. Sinon elle est appelée *irréductible*. Nous avons une condition spécifique pour tester la *réductibilité*.

Propriété 3. *Une fonction de coût non nulle* $f(X, \mathbf{Z}, Y)$ *est* réductible *si* $\exists x \in D_X, \exists z \in D_{\mathbf{Z}}$ *t.q.* $\min_{y \in D_Y} f(x, z, y) > 0$ *(i.e.* $f_1 = f[X, \mathbf{Z}] \neq 0$*) ou* $\exists y \in D_Y, \exists z \in D_{\mathbf{Z}}$ *t.q.* $\min_{x \in D_X} f(x, z, y) > 0$ *(i.e.* $f_2 = f[\mathbf{Z}, Y] \neq 0$*).*

Propriété 4. *Si* $f_1(X, \mathbf{Z})$, $f_2(\mathbf{Z}, Y)$, $\Delta(X, \mathbf{Z}, Y)$, *trois fonctions de coûts (positives), est une réduction maximale de* $f(X, \mathbf{Z}, Y)$ *alors* $\Delta(X, \mathbf{Z}, Y)$ *est irréductible.*

En utilisant la propriété 3, nous prouvons le lemme suivant.

Lemme 1. *Si* $f(X, \mathbf{Z}, Y)$ *une fonction de coûts non nulle est* irréductible *alors* $\exists z \in D_{\mathbf{Z}}, \exists k, l \in D_Y (k \neq l), \exists u, v \in D_X (u \neq v)$ *t.q.* $f(u, z, k) \boxminus f(u, z, l) \not\triangleq f(v, z, k) \boxminus f(v, z, l)$.

Preuve par l'absurde. Supposons $\forall z \in D_{\mathbf{Z}}, \forall k, l \in D_Y (k \neq l), \forall u, v \in D_X (u \neq v)$ tel que $f(u, z, k) \boxminus f(u, z, l) \triangleq f(v, z, k) \boxminus f(v, z, l)$. Nous rappelons que les fonctions de coûts ont leur coût dans $E^+ = \mathbb{N} \cup \{\top\}$ et non dans E. Nous avons :

Premier cas :

$\exists u, z, k, l$ **t.q.** $f(u, z, k) \boxminus f(u, z, l) \not\triangleq 0$

$\boxed{f(u, z, k) \boxminus f(u, z, l) > 0 \text{ ou } f(u, z, k) \boxminus f(u, z, l) = \top}$
Donc $0 \leqslant f(u, z, l) < f(u, z, k)$.
De plus $\forall v \in D_X$,
$f(u, z, k) \boxminus f(u, z, l) \triangleq f(v, z, k) \boxminus f(v, z, l)$.
Alors $\forall v \in D_X,\ f(v, z, k) \boxminus f(v, z, l) > 0$, ou $f(v, z, k) \boxminus f(v, z, l) = \top$, ou $f(v, z, k) \boxminus f(v, z, l) = \Omega$.
Par conséquent $0 \leqslant f(v, z, l) < f(v, z, k)$ ou $f(v, z, k) = f(v, z, l) = \top$. Donc $\min_{v \in D_X} f(v, z, k) > 0$.

$\boxed{f(u, z, k) \boxminus f(u, z, l) < 0 \text{ or } f(u, z, k) \boxminus f(u, z, l) = -\top}$
Donc $0 \leqslant f(u, z, k) < f(u, z, l)$.
De plus $\forall v \in D_X$,
$f(u, z, k) \boxminus f(u, z, l) \triangleq f(v, z, k) \boxminus f(v, z, l)$.
Ainsi $\forall v \in D_X,\ f(v, z, k) \boxminus f(v, z, l) < 0$, ou $f(v, z, k) \boxminus f(v, z, l) = -\top$, ou $f(v, z, k) \boxminus f(v, z, l) = \Omega$.
Par conséquent $0 \leqslant f(v, z, k) < f(v, z, l)$ ou $f(v, z, k) = f(v, z, l) = \top$. Donc $\min_{v \in D_X} f(v, z, l) > 0$.

Second cas :

$\forall u, z, k, l$ **t.q.** $f(u, z, k) \boxminus f(u, z, l) \triangleq 0$

$\boxed{f(u, z, k) \boxminus f(u, z, l) = \Omega}$
Donc $f(u, z, k) = f(u, z, l) = \top$.
De plus $\forall p \in D_Y,\ f(u, z, p) \boxminus f(u, z, l) = 0$ ou $f(u, z, p) \boxminus f(u, z, l) = f(u, z, p) \boxminus \top = \Omega$. Ainsi $f(u, z, p) = \top$. Donc $f(u, z, l) = f(u, z, k) = f(u, z, p) = \top$, et $\min_{p \in D_Y} f(u, z, p) = \top > 0$.

$\boxed{f(u, z, k) = f(u, z, l) > 0}$
Donc $f(u, z, k) \boxminus f(u, z, l) = 0$ et par hypothèse, $\forall p \in D_Y,\ f(u, z, p) \boxminus f(u, z, l) = 0$.
Ainsi $f(u, z, k) \boxminus f(u, z, l) = f(u, z, p) \boxminus f(u, z, l)$ et $0 < f(u, z, l) = f(u, z, k) = f(u, z, p)$.
Donc $\min_{p \in D_Y} f(u, z, p) > 0$.

$\boxed{f(u, z, k) = f(u, z, l) = 0}$
Par hypothèse, $\forall p \in D_Y,\ f(u, z, p) \boxminus f(u, z, l) \triangleq 0$.
Alors $f(u, z, p) \boxminus 0 \triangleq 0$. Ainsi $f(u, z, p) \boxminus f(u, z, l) \neq \Omega$ et donc $f(u, z, k) \boxminus f(u, z, l) = f(u, z, p) \boxminus f(u, z, l)$.
Finalement $\forall p \in D_Y, f(u, z, p) = 0$. Nous pouvons conclure en utilisant la propriété 3 que $f(X, \mathbf{Z}, Y)$ est réductible ou égal à la fonction nulle (si de manière récursive on est toujours dans le dernier sous-cas). $\square$

L'opération $a \boxplus b$ représente l'addition de deux éléments de $a, b \in E$: $a \boxplus b = a + b$ sauf

$\top \boxplus -\top = -\top \boxplus \top = a \boxplus \Omega = \Omega \boxplus b = \Omega$,
$\top \boxplus \top = \top \boxplus c = c \boxplus \top = \top$,
$-\top \boxplus -\top = -\top \boxplus c = c \boxplus -\top = -\top$ avec $c \in \mathbb{Z}$.

Maintenant nous pouvons prouver le théorème 2.

$\boxed{\Longrightarrow}$ Supposons $f(X, \mathbf{Z}, Y) = f_1(X, \mathbf{Z}) + f_2(\mathbf{Z}, Y)$ et $f_1(X, \mathbf{Z})$, $f_2(\mathbf{Z}, Y)$ fonctions positives ($0 \leqslant f_1(X, \mathbf{Z}) \leqslant f(X, \mathbf{Z}, Y)$). Alors les systèmes suivants $\mathcal{S}_z$ d'équations linéaires (utilisant $\boxplus$ et $\stackrel{\circ}{=}$ operateurs) ont une solution.

$$(\mathcal{S}_z) \begin{cases} f_1(1z) \boxplus f_2(z1) \stackrel{\circ}{=} f(1z1) \\ f_1(uz) \boxplus f_2(zk) \stackrel{\circ}{=} f(uzk) \quad \forall u, k \\ f_1(d_X z) \boxplus f_2(z d_Y) \stackrel{\circ}{=} f(d_X z d_Y) \end{cases}$$

$\forall z \in D_{\mathbf{Z}}$, $\forall k, l \in D_Y$, $\forall u \in D_X$, de $\mathcal{S}_z$, nous déduisons

$$\begin{aligned} f(uzk) \boxminus f(uzl) &\stackrel{\circ}{=} (f_1(uz) \boxplus f_2(zk)) \boxminus (f_1(uz) \boxplus f_2(zl)) \\ &\stackrel{\circ}{=} (f_1(uz) \boxminus f_1(uz)) \boxplus (f_2(zk) \boxminus f_2(zl)) \end{aligned}$$

1^{er} **cas :** $f_1(uz) \in \mathbb{N}$
$f(uzk) \boxminus f(uzl) \stackrel{\circ}{=} (f_1(uz) \boxminus f_1(uz)) \boxplus (f_2(zk) \boxminus f_2(zl)) \stackrel{\circ}{=} 0 \boxplus (f_2(zk) \boxminus f_2(zl)) \stackrel{\circ}{=} f_2(zk) \boxminus f_2(zl)$

2^{nd} **cas :** $f_1(uz) = \top$
$f(uzk) \boxminus f(uzl) \stackrel{\circ}{=} (f_1(uz) \boxminus f_1(uz)) \boxplus (f_2(zk) \boxminus f_2(zl)) \stackrel{\circ}{=} \Omega \boxplus (f_2(zk) \boxminus f_2(zl)) \stackrel{\circ}{=} \Omega$

Nous pouvons conclure que $\forall z \in D_{\mathbf{Z}}, \forall k, l \in D_Y$ $(k \neq l) \stackrel{\circ}{=}_{u \in D_X} f(uzk) \boxminus f(uzl)$

$\boxed{\Longleftarrow}$ Supposons $\forall z \in D_{\mathbf{Z}}, \forall k, l \in D_Y$ $(k < l), \forall u, v \in D_{\mathbf{X}}$: $f(uzk) \boxminus f(uzl) \stackrel{\circ}{=} f(vzk) \boxminus f(vzl)$.
De plus, nous avons $f(X, \mathbf{Z}, Y) = f_1(X, \mathbf{Z}) + f_2(\mathbf{Z}, Y) + \Delta(X, \mathbf{Z}, Y)$ $f_1(X, \mathbf{Z})$, $f_2(\mathbf{Z}, Y)$ et $\Delta(X, \mathbf{Z}, Y)$ des fonctions positives qui définissent les systèmes linéaires suivants :

$$(\mathcal{S}'_z) \begin{cases} f_1(1z) \boxplus f_2(z1) \boxplus \Delta(1z1) \stackrel{\circ}{=} f(1z1) \\ f_1(uz) \boxplus f_2(zk) \boxplus \Delta(uzk) \stackrel{\circ}{=} f(uzk) \quad \forall u, k \\ f_1(d_X z) \boxplus f_2(z d_Y) \boxplus \Delta(d_X z d_Y) \stackrel{\circ}{=} f(d_X z d_Y) \end{cases}$$

$$\begin{aligned} & f(uzk) \boxminus f(uzl) \\ & \stackrel{\circ}{=} (f_1(uz) \boxplus f_2(zk) \boxplus \Delta(uzk)) \boxminus (f_1(uz) \boxplus f_2(zl) \boxplus \Delta(uzl)) \\ & \stackrel{\circ}{=} f_1(uz) \boxplus f_2(zk) \boxplus \Delta(uzk) \boxplus (-f_1(uz)) \boxplus (-f_2(zl)) \\ & \boxplus (-\Delta(uzl)) \\ & \stackrel{\circ}{=} (f_1(uz) \boxminus f_1(uz)) \boxplus (f_2(zk) \boxminus f_2(zl)) \boxplus (\Delta(uzk) \boxminus \Delta(uzl)) \end{aligned}$$

Supposons que $f(X, \mathbf{Z}, Y)$ ne se décompose pas en $\{f_1(X, \mathbf{Z}), f_2(\mathbf{Z}, Y)\}$, avec la propriété 4 nous pouvons trouver $\Delta(X, \mathbf{Z}, Y)$ tel que $\Delta(X, \mathbf{Z}, Y)$ est une fonction de coût non nulle irréductible. Avec le théorème 1 nous avons $\exists z \in D_{\mathbf{Z}}$, $k, l \in D_Y, k < l$ et $u, v \in D_X, u < v$ t.q. $\Delta(uzk) \boxminus \Delta(uzl) \not\stackrel{\circ}{=} \Delta(vzk) \boxminus \Delta(vzl)$.

Nous en déduisons que $\Delta(uzk) \boxminus \Delta(uzl) \neq \Omega$ et $\Delta(vzk) \boxminus \Delta(vzl) \neq \Omega$ avec la définiton de $\stackrel{\circ}{=}$.
Nous avons également $f(uzk) \boxminus f(uzl) \neq \Omega \neq f(vzk) \boxminus f(vzl)$. Alors $f(uzk) \boxminus f(uzl) \stackrel{\circ}{=} f(vzk) \boxminus f(vzl) \stackrel{\circ}{=} \varphi$ t.q. $\varphi \in \mathbb{Z} \cup \{\top, -\top\}$ donc $f_1(uz) \boxminus f_1(uz) = 0 = f_1(vz) \boxminus f_1(vz)$ et $f_2(zk) \boxminus f_2(zl) \neq \Omega$

En utilisant les propriétés précédentes :

$$\begin{aligned} & \Delta(uzk) \boxminus \Delta(uzl) \neq \Delta(vzk) \boxminus \Delta(vzl) \\ & f_2(zk) \boxminus f_2(zl) \boxplus \Delta(uzk) \boxminus \Delta(uzl) \\ & \quad \neq f_2(zk) \boxminus f_2(zl) \boxplus \Delta(vzk) \boxminus \Delta(vzl) \\ & f_1(uz) \boxminus f_1(uz) \boxplus f_2(zk) \boxminus f_2(zl) \boxplus \Delta(uzk) \boxminus \Delta(uzl) \\ & \quad \neq f_1(vz) \boxminus f_1(vz) \boxplus f_2(zk) \boxminus f_2(zl) \boxplus \Delta(vzk) \boxminus \Delta(vzl) \\ & f_1(uz) \boxplus f_2(zk) \boxplus \Delta(uzk) \boxminus (f_1(uz) \boxplus f_2(zl) \boxplus \Delta(uzl)) \\ & \quad \neq f_1(vz) \boxplus f_2(zk) \boxplus \Delta(vzk) \boxminus (f_1(vz) \boxplus f_2(zl) \boxplus \Delta(vzl)) \\ & f(uzk) \boxminus f(uzl) \neq f(vzk) \boxminus f(vzl) \end{aligned}$$

C'est en contradiction avec la première hypothèse : $\forall\ k, l \in D_Y$, $\forall\ z \in D_{\mathbf{Z}}$ $f(uzk) \boxminus f(uzl) \stackrel{\circ}{=} f(vzk) \boxminus f(vzl)$.

C Preuve du Théorème 3

Lemme 2. *Soit une fonction de coûts $f(X, \mathbf{Z}, Y)$ décomposable par paire par rapport à X, Y.*
Si $f_1(x, z) = \min_{y \in D_Y} f(x, z, y)$ alors $\forall z \in D_{\mathbf{Z}}$ $\exists k \in D_Y$ tel que $\forall x \in D_X$, $f_1(x, z) = f(x, z, k)$.

Démonstration. Soit $z \in D_{\mathbf{Z}}$.

$\boxed{\exists x \in D_x, \exists y,\ f(x, z, y) \neq \top}$
Soit $D_K = \{k_1, k_2, \ldots, k_{d_Y}\}$ tel que $\forall i < j$, $f(x, z, k_i) \leqslant f(x, z, k_j)$. Nous en déduisons $\forall i \in [1, d_Y]$, $f(x, z, k_1) \leqslant f(x, z, k_i)$ et $f_1(x, z) = \min_{y \in D_Y} f(x, z, y) = f(x, z, k_1)$.
En utilisant le théorème 2, nous trouvons $\forall k_p \in D_K$ $\forall u \in D_X$, $0 \leqslant f(x, z, k_p) \boxminus f(x, z, k_1) \stackrel{\circ}{=} f(u, z, k_p) \boxminus f(u, z, k_1)$. Donc $f(u, z, k_1) \leqslant f(u, z, k_p)$ ou $f(u, z, k_1) = f(u, z, k_p) = \top$, ainsi $\forall u \in D_X, f_1(x, z) = \min_{y \in D_Y} f(u, z, y) = f(u, z, k_1)$.

$\boxed{\forall x \in D_x, \forall y,\ f(x, z, y) = \top}$
$\forall x \in D_X$, $f_1(x, z) = \min_{y \in D_Y} f(x, z, y) = \top$, donc soit $k \in D_y$, $\forall x \in D_X$, $f_1(x, z) = f(x, z, k) = \top$ □

Maintenant nous pouvons prouver le théorème 3.
Nous avons $f_1(x, z) = \min_{y \in D_Y} f(x, z, y)$ et $f_2(z, y) = \min_{x \in D_X} [f(x, z, y) - f_1(x, z)]$ donc $f_2(z, y) \leqslant f(x, z, y) - f_1(x, z)$.

Si $f_2(z, y) = f(x, z, y) - f_1(x, z)$, $f(x, z, y) = f_1(x, z) + f_2(z, y)$.

Si $f_2(z, y) < f(x, z, y) - f_1(x, z)$,

$\boxed{f(x, z, y) = f_1(x, z) = \top}$
Nous avons $f_1(x, z) + \min_{x \in D_X} [f(x, z, y) - f_1(x, z)] = \top = f(x, z, y)$ donc$f(x, z, y) = f_1(x, z) + f_2(z, y)$.

$\boxed{f(x, z, y) \neq \top \textbf{ ou } f_1(x, z) \neq \top}$
Nous avons $\min_{u \in D_X} [f(u, z, y) - f_1(u, z)] < f(x, z, y) - f_1(x, z)$ donc $f(u, z, y) - f_1(u, z) < f(x, z, y) - f_1(x, z)$ avec $u = \text{argmin}_{u \in D_X} [f(u, z, y) - f_1(u, z)]$. En utilisant le lemme 2, nous avons $l = \text{argmin}_{k \in D_Y} f(x, z, k) = \text{argmin}_{k \in D_Y} f(u, z, k)$,

ainsi $f(u,z,l) - f(u,z,l) < f(x,z,y) - f(x,z,l)$, mais également $f(u,z,y) \boxminus f(u,z,l) \doteq f(x,z,y) \boxminus f(x,z,l)$ car f est décomposable par paire par rapport à X, Y. De plus, $f(x,z,y) \neq \top$ ou $f(x,z,l) \neq \top$, ainsi $f(u,z,y) - f(u,z,l) = f(x,z,y) - f(x,z,l)$. Finalement $f_2(z,y) = f(x,z,y) - f_1(x,z)$, ainsi $f(x,z,y) = f_1(x,z) + f_2(z,y)$.

Références

[1] M. Cooper, S. de Givry, M. Sanchez, T. Schiex, M. Zytnicki, and T. Werner. Soft arc consistency revisited. *Artificial Intelligence*, 174(7–8) :449–478, 2010.

[2] S. de Givry, J. Larrosa, P. Meseguer, and T. Schiex. Solving max-sat as weighted csp. In *Proc. of CP-03*, pages 363–376, Kinsale, Ireland, 2003.

[3] M Fishelson, N Dovgolevsky, and D Geiger. Maximum likelihood haplotyping for general pedigrees. *Human Heredity*, 59 :41–60, 2005.

[4] J Hammersley and P Clifford. Markov fields on finite graphs and lattices, 1971.

[5] D. Heckerman and J. Breese. Causal independence for probability assessment and inference using bayesian networks. *IEEE Systems, Man, and Cyber.*, 26(6) :826–831, 1996.

[6] D. Koller and N. Friedman. *Probabilistic Graphical Models*. The MIT Press, 2009.

[7] J. Larrosa. Boosting search with variable elimination. In *CP*, pages 291–305, Singapore, 2000.

[8] S. Lauritzen. *Graphical Models*. Oxford University Press, 1996.

[9] C. Lecoutre, L Saïs, S. Tabary, and V. Vidal. Reasoning from last conflict(s) in constraint programming. *Artificial Intelligence*, 173 :1592,1614, 2009.

[10] R. Marinescu and R. Dechter. Memory intensive and/or search for combinatorial optimization in graphical models. *Artificial Intelligence*, 173(16-17) :1492–1524, 2009.

[11] I. Meiri, R. Dechter, and J. Pearl. Tree decomposition with applications to constraint processing. In *Proc. of AAAI'90*, pages 10–16, Boston, MA, 1990.

[12] P. Meseguer, F. Rossi, and T. Schiex. Soft constraints processing. In *Handbook of Constraint Programming*, chapter 9. Elsevier, 2006.

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

[14] M. Sánchez, P. Meseguer, and J. Larrosa. Using constraints with memory to implement variable elimination. In *ECAI*, pages 216–220, Spain, 2004.

[15] P. Savicky and J. Vomlel. Exploiting tensor rank-one decomposition in probabilistic inference. *Kybernetika*, 43(5) :747–764, 2007.

Modèles en Iles Dynamiques

Adrien Goëffon et Frédéric Lardeux

LERIA - Université d'Angers
{prenom.nom}@univ-angers.fr

Résumé

Dans ce papier, nous proposons un cadre de travail pour les modèles en iles dont les topologies sont représentées par des graphes complets étiquetés. À chaque arc est associée une probabilité de migration entre deux iles. Nous montrons ici comment faire évoluer ces probabilités dynamiquement au cours de la recherche en fonction de l'impact des dernières migrations, ce qui permet d'améliorer le paramétrage et l'efficacité des algorithmes génétiques et évolutionnaires. Deux types d'application de ce cadre de travail sont détaillés ici : des modèles en iles auto-adaptatifs et des algorithmes évolutionnaires autonomes.

Abstract

In this paper, we present a framework for island-based evolutionary algorithms, using a complete graph modeling. Each directed edge is labeled by a migration probability between two islands. We show that probabilities can evolve dynamically during the search with respect to the last migrations impact, in order to improve the efficiency of evolutionary algorithms and to make their tuning easier. Two applications of this framework are detailed here : auto-adaptive island models and autonomous evolutionary algorithms.

1 Introduction

Les algorithmes génétiques (AG) [10] sont largement utilisés pour la résolution approchée de problème NP-difficiles. Ils sont relativement simples à concevoir et peuvent produire sur de nombreux problèmes d'optimisation de très bons résultats en terme de qualité des solutions et de robustesse de l'algorithme. Néanmoins, leur efficacité dépend d'un certain nombre de facteurs, de la représentation des configurations [18] à la fonction de fitness utilisée [29], en passant par les opérateurs de mutation et de croisement définis [21] ou encore des paramétrages plus généraux (taille de la population, fréquence des mutations, contrôle de la diversité, sélections des individus, élitisme, *etc.*) [12, 11, 25]. Même avec un effort particulier pour adapter un AG efficace à un problème particulier, on observe rapidement sur certaines instances plus critiques ou difficiles un certain nombre de limitations en terme de performances générales ou de passage à l'échelle de l'algorithme.

Afin de rendre les AG plus efficaces et puissants, les techniques habituelles consistent à les hybrider avec des algorithmes de recherche locale (algorithmes mémétiques [19]), et/ou d'utiliser des modèles en ile permettant à la fois de combiner plus efficacement l'intensification et la diversification de la recherche, mais aussi de rendre l'algorithme plus facilement parallélisable [28].

Depuis une vingtaine d'années et l'avènement des premiers algorithmes évolutionnaires distribués [26], les modèles en ile sont de plus en plus étudiés dans la communauté évolutionnaire. La principale problématique est de définir à la fois la topologie du modèle (*où peut migrer tel individu ?*) et les politiques de migration (*quel individu doit migrer ?*) de manière à ralentir la convergence générale de la population tout en favorisant le brassage global des bons individus. Araujo *et al.* ont dressé dans [2] un solide état de l'art des modèles en ile, concernant notamment la question des politiques de migration. On peut observer qu'un nombre important de topologies (on retrouve dans [14, 22] de nombreux types de modèles comme les chaines, anneaux, hypercubes et bien d'autres) et de politiques [3, 6, 27, 5, 7, 1] ont été définies, grandement basées sur des mesures de diversité locales et globales (par population / sur l'ensemble des iles). Dans tous les cas, la fréquence et la quantité des migrations demeure difficile à déterminer [24], d'autant plus que les interactions entre les deux niveaux d'évolution (intra- et inter-iles) sont très importantes [23]. Idéalement, une stratégie d'évolution globale intelligente devrait tirer un avantage de ces interactions et maximiser le bénéfice des migrations. Mais à partir de simples indicateurs intra- et inter-iles (typiquement des mesures de diversité et de fitness), il est difficile de prédire à quel moment des individus doivent migrer, lesquels, où

et pour quel impact.

Ces considérations nous ont motivé à développer un modèle en ile dynamique dont le but est d'auto-adapter sa topologie et ses politiques de migration au cours de la recherche en fonction d'indicateurs définis en paramètre. Cela peut être les propriétés des sous-populations, mais surtout et dans le cadre d'une approche originale et plus efficace, les effets des précédentes migrations. La première application est donc de définir dynamiquement une politique optimale de croisements entre individus dans les AG pour une meilleure efficacité. Dans un second temps, nous étendons le champ d'application de notre modèle dynamique, non plus pour optimiser les croisements entre individus, mais pour sélectionner automatiquement, dans le cadre d'un algorithme évolutionnaire, les opérateurs de recherche locale les plus pertinents au cours de la recherche. Dans cette extension, les iles ne constituent plus qu'un simple partitionnement entre les individus, mais disposent chacunes de leur propres règles et paramétrages, par exemple un opérateur de recherche locale différent.

Ce papier est structuré en trois parties. Dans la section 2, nous décrivons de manière générale le modèle en ile dynamique, en le présentant comme un cadre de travail. La section 3 se focalise sur l'application principale d'un tel modèle, à savoir la modification dynamique de sa topologie [16]. Des résultats sur les problèmes 0/1KP et MAX-SAT montreront le bien-fondé de cette approche. Dans la section 4, nous montrons comment une telle modélisation peut être utilisée pour définir un algorithme évolutionnaire autonome, choisissant par exemple les opérateurs de recherche locale les plus pertinents au cours de la recherche.

2 Généralisation des modèles en iles

2.1 Cadre de travail

Comme le rappellent les auteurs de [2], plusieurs paramètres définissent un modèle en iles :

- le nombre d'individus pouvant migrer,
- la politique de sélection des individus migrants,
- la politique de remplacement des individus migrants,
- la topologie de la communication entre sous-populations, et
- la nature synchrone ou asynchrone de la connexion entre sous-populations.

Nous proposons ici une modélisation qui généralise tous ces paramètres, et tout en permettant de rendre le modèle en iles dynamique.

Nous représentons ce dernier par un graphe où les nœuds représentent les iles (les sous-populations) et les arcs les possibilités de migrations. Chaque arc possède une valeur qui représente la probabilité pour un individu de migrer d'une ile vers une autre. L'auto-adaptation de ce modèle est gérée à l'aide d'un mécanisme de récompense/pénalité. Les probabilités de migration (valeurs des arcs) sont mises à jour après chaque cycle de migration en fonction des derniers effets de la migration. Si l'ile qui accueille un individu voit sa population améliorée (resp. détériorée) alors la probabilité de migration correspondante est augmentée (resp. diminuée). Ici, la qualité des populations est évaluée en terme de fitness moyenne. diversity of the population if the modeling imposes it.

Le contrôle dynamique des paramètres (comme le taux de migration) rend possible de manipuler des iles de différentes tailles (sauf si nous décidons de l'interdire). Ce mécanisme empêche les sous-populations de faible qualité de dépenser autant d'effort de calcul que celles de bonne qualité et permet de gérer la fusion des populations. Si chaque ile est associée à un opérateur de mutation différent, un effort de recherche locale différent ou à une paramétrisation locale, alors l'algorithme pourra dynamiquement fournir une répartition bien adaptée des individus durant tout le processus de recherche.

2.2 Mise à Jour Dynamique du Modèle

La figure 1 est un exemple de modèle avec trois iles ($i1$, $i2$ et $i3$). La figure 1.a représente les iles avec leurs individus ainsi que les valeurs de migration (probabilités) d'une ile à une autre. Dans la figure 1.b, la destination de chaque individu est choisie en fonction des valeurs des probabilités de migration. La plupart des individus restent sur la même ile (les valeurs des boucles sont généralement plus élevées) mais plusieurs individus migrent vers d'autres iles (deux individus vont de $i1$ à $i2$ et un de $i3$ à $i2$). Après ces migrations, sur chaque ile, les opérateurs tels que le croisement, la mutation ou la recherche locale sont appliqués sur les individus. On met ensuite à jour les valeurs de migration en fonction de la provenance des parents dont les fils (*i.e.* individus obtenus par croisement) améliorent (resp. détériorent) la population. Pour chaque parent, la valeur de l'arc entre la dernière ile qu'il a visitée et son ile actuelle est augmentée (resp. diminuée) pour prendre en compte l'impact de la migration. Par exemple, sur la figure 1.c, si nous observons seulement $i2$ et que la valeur de récompence/pénalité est fixée à 5 points (± 0.05), plusieurs valeurs sont mises à jour :

- ($i1 \rightarrow i2$) diminue de 0.50 à 0.40 car deux individus venant de $i1$ ont produit un individu fils détériorant la population de $i2$;
- ($i2 \rightarrow i2$) diminue de 0.95 à 0.85 car les individus de l'ile $i2$ n'améliorent pas la population de $i2$;
- ($i3 \rightarrow i2$) augmente de 0.20 à 0.30 car l'individu venant de $i3$ a produit un individu fils améliorant la population de $i2$;
- dû à la normalisation, d'autres valeurs en relation avec $i2$ doivent être ajustées.

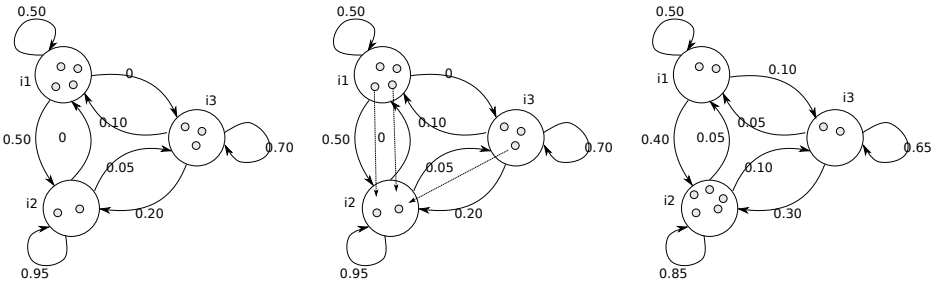


FIGURE 1 – Communication entre les sous-populations avec une représentation en graphe complet

3 Modèles en iles auto-adaptatifs

Dans cette section nous proposons de mesurer l'efficacité générale du schéma dynamique de modèle en iles (MID), appliqué à deux problèmes d'optimisation classiques : le problème du sac à dos 0/1 (0/1 KP) et le problème MAX-SAT. Pour cette étude, l'objectif principal n'est pas de proposer un algorithme puissant qui égale les meilleures méthodes connues mais plutôt de mesurer la pertinence d'un tel modèle. C'est pour cette raison que nous avons déconnecté les opérateurs de recherche locale et que nous nous intéressons uniquement à un AG classique partitionné en iles.

Pour ces deux problèmes, nous comparons les performances de quatre configurations basiques d'algorithmes génétiques :

- Un AG classique d'une seule ile (AG),
- Un algorithme MID standard ($\mathrm{MID}_{\mathrm{stand}}$),
- Un aglorithme MID paramétré avec une topologie en anneau unidirectionnel $\mathrm{MID}_{\mathrm{ring}}$ qui simule un modèle en ile classique avec des migrations circulaires (à chaque processus de migration, les meilleurs individus migrent vers les populations suivantes),
- Un AG parallèle ($\mathrm{AG}/\!/$) avec plusieurs iles mais pas de migration.

Notons que l'AG classique correspond à un modèle en iles où l'intervalle des migrations est minime tandis, que l'AG parallèle est un modèle en iles sans aucune migration (ou avec un intervalle de migration infini).

3.1 Paramétrage des algorithmes génétiques

Les quatre configurations des AG ont deux types de caractéristiques. Toutes les valeurs numériques sont empiriquement obtenues et confirmées par la méthode REVAC [20].

1. Caractéristiques intra-iles :
   - type d'AG : steady-state
   - élitisme : oui
   - sélection : tournoi
   - mutation : aléatoire sur les fils avec une probabilité de 0.5
   - croisement : uniforme
2. Caractéristiques inter-iles :
   - nombre d'iles : 20
   - nombre total d'individus : 600
   - répartition de départ : équilibré (30 individus par ile)
   - nombre total de croisements : 216000 (360×600 individus)
   - probabilité initiale de migration : voir section suivante
   - récompense : 5 points
   - penalité : 5 points

Le nombre de croisements pour une ile entre chaque migration est proportionelle à son nombre d'individus. Ce choix assure le même taux de croisements pour chaque ile quelle que soit sa taille.

Probabilités initiales de migration Pour donner la même puissance d'attractivité à chaque ile, les probabilités initiales de migrations doivent être symétriques. Au départ, la probabilité de rester sur la même ile est beaucoup plus élevée que celle de migrer vers une autre ile afin d'exploiter les populations initiales. Par exemple, la matrice initiale correspondant à un modèle en iles avec trois iles pourrait être le suivant :

		Destination		
		$i1$	$i2$	$i3$
Source	$i1$	0.75	0.125	0.125
	$i2$	0.125	0.75	0.125
	$i3$	0.125	0.125	0.75

3.2 Conditions expérimentales

Les algorithmes utilisés dans nos expériences sont appliqués 10 fois pour chaque instance. Pour s'assurer que la différence de comportement n'est pas due à la population initiale, le même jeu de 10 graines aléatoires sont utilisées

par tous les algorithmes. Les résultats présentés dans les tables sont des moyennes. Les écarts-types étant tous très faibles, ils ne sont pas reportés spécifiquement.

3.3 Problème du sac à dos 0/1

Le problème du sac à dos 0/1 est un problème combinatoire bien connu. Ayant n objets dont le poids w_i et la valeur v_i sont connus ($x_i \in \{1, \ldots, n\}$), le but est de trouver un sous-ensemble d'objets de valeur maximale tel que le poids total ne dépasse pas une capacité donnée W. Dans le problème binaire 0/1, chaque objet ne peut être sélectionné qu'une seule fois ($x_i \in \{0, 1\}$, où x_i est le nombre de copies sélectionnées pour l'objet i).

Plus précisément, le problème du sac à dos 0/1 est formulé comme un problème d'optimisation par :

$$\text{maximise} \sum_{i=1}^{n} v_i x_i, \text{ t.q.} \sum_{i=1}^{n} w_i x_i \leq W, \; x_i \in \{0, 1\}$$

Pour plus d'informations sur ce problème, nous invitons le lecteur à se référer à [15].

Les algorithmes à base d'iles $\mathrm{MID}_{\mathrm{stand}}$ et $\mathrm{MID}_{\mathrm{ring}}$ ainsi que les autres algorithmes $\mathrm{GA}_{\mathrm{classic}}$ et AG$/\!/$ ont été testés sur cinq instances du problème. Ces instances ont été générées en se basant sur la définition donnée par [4] et le générateur proposé dans [17] avec les paramètres suivants :

- nombre d'objets $\in \{100, 250, 500, 1000, 2000\}$
- coefficient : 10000
- type : avis subset-sum
- nombre de tests en séries : 1000

Les expériences ont montré que seules les trois dernières instances (celles avec 500, 1000 et 2000 objets) sont représentatives pour comparer les algorithmes. Les deux premières ne présentent pas de difficulté particulière et les résultats sont donc similaires quel que soit l'algorithme. Nous nous focaliserons donc sur les trois instances aléatoires : n500, n1000 et n2000.

La table 1 montre l'efficacité de chaque méthode sur ces intances. Il n'est pas surprenant que le modèle en iles dynamique soit plus performant que les AG traditionnels. Cependant, les différences de performances sont assez importantes sachant que les dernières améliorations sont souvent plus dures à trouver pour le problème du sac à dos. Un point intéressant dans cette expérience est que le schéma de rotation classique ($\mathrm{MID}_{\mathrm{ring}}$) n'est pas compétitif ; comparativement, l'AG classique fournit souvent de meilleurs résultats pour les deux instances les plus difficiles. La principale raison provient probablement de la taille relativement petite des iles (20 individus) qui est adaptative au cours du processus de recherche pour $\mathrm{MID}_{\mathrm{stand}}$ alors qu'elle reste inchangée pour $\mathrm{MID}_{\mathrm{ring}}$.

3.3.1 Le Problème MAX-SAT

Pour une formule booléenne donnée en CNF (conjonctions de clauses qui sont des disjonctions de littéraux), le problème MAX-SAT consiste à fournir une affectation aux variables booléennes telle que le nombre de clauses vraies soit maximum. La formule est satisfiable si et seulement s'il existe une affectation qui rend vraies toutes les clauses.

Trois instances satisfiables sont utilisées pour nos expériences :

- f600 : instance aléatoire avec 600 variables et 2550 clauses,
- f1000 : instance aléatoire avec 1000 variables et 4250 clauses, et
- qg1-7.suffled : instance de carré latin avec 686 variables et 6816 clauses.

Dans la table 2 nous observons que $\mathrm{MID}_{\mathrm{stand}}$ fournit les meilleurs résultats pour les trois instances. AG$/\!/$ est le moins performant et AG obtient des résultats légèrement moins bons que $\mathrm{MID}_{\mathrm{ring}}$. Les résultats pour $\mathrm{MID}_{\mathrm{stand}}$ et $\mathrm{MID}_{\mathrm{ring}}$ sont calculés avec la meilleure fréquence de migration empiriquement trouvée (voir ci-après).

3.3.2 Discussion

Comme nous l'avons vu précédemment, $\mathrm{MID}_{\mathrm{stand}}$ obtient des résultats encourageants comparativement aux autres modèles d'algorithmes génétiques testés. Pourtant, la différence entre $\mathrm{MID}_{\mathrm{stand}}$ et $\mathrm{MID}_{\mathrm{ring}}$ provient uniquement du type de migration, tandis que les différences entre AG$/\!/$, AG et $\mathrm{MID}_{\mathrm{stand}}$ se fait sur la fréquence des migrations. Cette fréquence, ou plutôt période dans ce cas, est donnée par un nombre moyen de croisements par individu. En conséquence, le nombre de croisements entre deux phases de migrations diffère d'une ile à l'autre et dépend de leur taille (*i.e.* de leur nombre d'individus).

L'algorithme génétique classique AG peut être considéré comme un modèle en ile avec une période de migration très faible ; l'algorithme parallèle AG$/\!/$ aurait lui une période de migration maximale, telle qu'il n'y ait aucune migration durant l'intégralité de la recherche (on rappelle qu'une périodicité faible correspond à une fréquence élevée). Entre ces deux algorithmes, un modèle en ile dynamique peut utiliser des fréquences différentes, qui engendreront des comportements différents. On peut observer figure 2 l'impact des fréquences de migrations sur l'ensemble des instances 0/1 KP et MAX-SAT mentionnées dans cet article.

Il est intéressant de noter que, lorsque la période est supérieure à 8 (*i.e.* 8 croisements par individu en moyenne sur chaque ile), $\mathrm{MID}_{\mathrm{stand}}$ et $\mathrm{MID}_{\mathrm{ring}}$ fournissent des résultats équivalents. Une explication possible est que la population a le temps de converger suffisamment avant la première phase migratoire. En effet, dans nos expérimentations, 240 croisements (période : 8, individus par ile : 30

Instance	AG	$\mathrm{MID}_{\mathrm{stand}}$	$\mathrm{MID}_{\mathrm{ring}}$	AG$/\!/$
n500	755 626.54	**760 620.50**	755 818.38	748 230.25
n1000	1 485 393.86	**1 502 549.22**	1 483 248.36	1 465 757.47
n2000	2 866 752.92	**2 910 891.46**	2 853 072.50	2 832 290.37

TABLE 1 – Comparaisons entre MID et des AG classiques

Instance	Nb Clauses	AG	$\mathrm{MID}_{\mathrm{stand}}$	$\mathrm{MID}_{\mathrm{ring}}$	AG$/\!/$
f600	2550	2513.80	**2533.40**	2518.70	2357.20
f1000	4250	4174.20	**4208.90**	4174.10	3890.50
qg1-7.shuffled	6816	6756.30	**6787.00**	6776.50	6211.70

TABLE 2 – Comparaison entre MID et les AG classiques

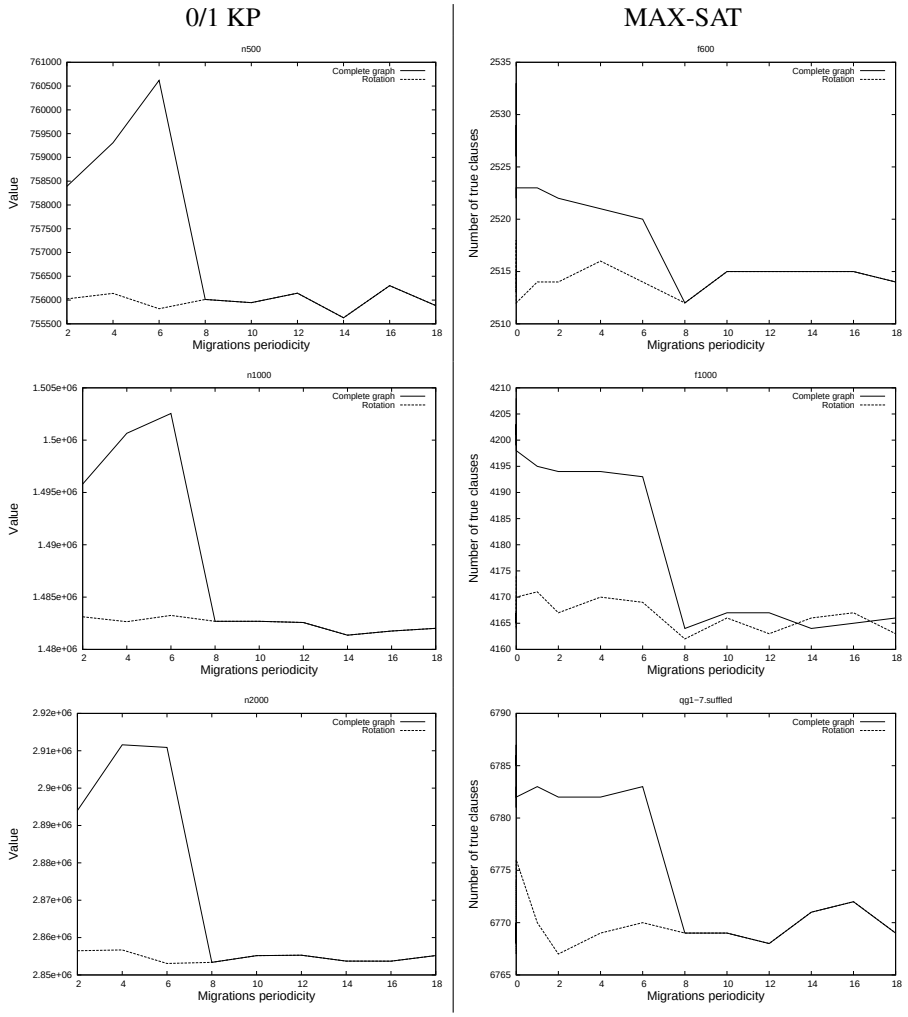


FIGURE 2 – Puissance de l'algorithme MID en fonction de la fréquence de migration (période élevée = migrations plus espacées)

au départ) sont appliqués sur la population de chaque ile avant la première migration. Si chaque population contient seulement des clones ou des individus très proches après cette première phase évolutionnaire, alors un schéma rotatif des migrations produira le même effet réduit qu'un modèle en ile basé sur un graphe complet. Précisons que

pour cette étude, nous avons délibérément retiré tout contrôle de diversité au sein des populations, afin d'observer plus précisément le comportement des différents modèles. C'est pour cette raison que les scores atteints sont moins bons que ceux du tableau 2.

Une autre observation est la faible variabilité des performances de $\mathrm{MID}_{\mathrm{ring}}$ lorsque l'on fait varier la périodicité des migrations. Hormis pour l'instance `qg1-7.suffled`, où les résultats sont meilleurs avec une faible périodicité, ce paramètre ne semble pas déterminant dans le cadre d'un schéma rotatif des migrations.

Avec une périodicité des migrations inférieure à 8, le schéma en graphe complet est clairement meilleur que le classique schéma rotatif. Pour les instances du 0/1 KP, une valeur fixée à 6 semble être la plus appropriée dans notre étude, tandis qu'une valeur plus faible fournira de meilleurs résultats pour les instances MAX-SAT. En fonction des problèmes, des instances ou des algorithmes évolutionnaires utilisés, une plus grande vitesse de convergence des individus profitera de phases migratoires plus rapprochées, mais une simple régulation de la convergence réduira ces différences d'efficacité. Il est également possible d'auto-réguler le rythme des fréquences migratoires, soit explicitement en faisant varier le paramètre durant la recherche à l'aide d'un contrôleur adapté, soit implicitement dans la définition des récompenses et pénalités. En effet, en renforçant ou relâchant les probabilités de migration des boucles (probabilité qu'un individu reste sur son ile lors d'une phase migratoire), on obtient un plus faible ou plus fort taux de migration dans le modèle.

4 Algorithmes évolutionnaires autonomes

Dans la section précédente, on présentait une évolution dynamique de la topologie du modèle, ce qui a pour effet de déterminer à tout moment des probabilités de migration pertinentes entre les différentes populations, dans le cadre d'un algorithme génétique classique partitionné.

Dans cette section, nous utiliserons le même cadre de travail pour sélectionner de manière autonome les opérateurs de recherche locale au sein d'un algorithme évolutionnaire classique. Pour mesurer de manière précise les apports de ce modèle, nous l'utiliserons au sein d'un algorithme de recherche locale à population et sans croisement. À chaque ile sera associé un opérateur de recherche locale particulier. L'objectif est ici non pas d'anticiper les meilleurs croisements entre les individus, mais de détecter à tout moment de la recherche le ou les opérateurs de recherche locale le(s) plus pertinent(s). Cette utilisation du cadre de travail constitue une approche originale dans la définition d'algorithmes autonomes [13, 8].

La mise à jour dynamique des probabilités de migrations entre les iles suivra le même modèle que dans la section 2.2, à ceci près qu'il faut estimer la pertinence des opérateurs de recherche locale et non des croisements. Dans la section précédente, nous évaluions les fils des individus immigrants afin de récompenser ou pénaliser les différents flux migratoires entrants. Dans cette section, le poids des arcs sera cette fois impacté en fonction du bénéfice obtenu du point de vue de l'individu immigrant (sur lequel l'opérateur de recherche locale propre à l'ile aura été appliqué). C'est-à-dire qu'à chaque stade de la recherche, les arcs entrants des iles comportant les meilleurs opérateurs de recherche locale vont se renfoncer et ainsi attirer une plus grande quantité d'invividus, ce qui va accélérer la recherche d'un point de vue global. Afin de rendre chaque opérateur toujours accessible (certains opérateurs seront plus efficaces en fin de recherche qu'au début, et réciproquement), il est impératif soit d'imposer un poids minimal à chaque arc du modèle, soit d'introduire un bruit dans la mise à jour des poids. Ces condisérations seront prises en compte lors du paramétrage du modèle, comme le fait de rendre l'algorithme plus ou moins intensif et réactif, à la manière des algorithmes de colonies de fourmis.

Nous proposons de mesurer l'intérêt d'une telle approche en l'appliquant au problème ONE-MAX. Il s'agit d'un problème très simple, dont on connait théoriquement les meilleurs opérateurs de recherche locale à appliquer en fonction de l'avancée de la recherche.

4.1 Le Problème ONE-MAX

Le problème ONE-MAX consiste à retrouver plus ou moins à l'aveugle une chaine de bits ayant un maximum de 1. Notre problématique consiste à débuter la recherche avec une chaine de bits initialisée par une suite de 0 pour la terminer avec une suite de 1, en utilisant uniquement des séries de flips aléatoires. En début de recherche, les meilleurs opérateurs de recherche locale sont ceux permettant de *flipper* un maximum de bits, tandis qu'en fin de recherche, les opérateurs effectuant un nombre réduit de flips sont censés être les plus efficaces.
Concrètement, nous allons travailler avec une chaine de l bits (dans notre expérience 1000) et nous utiliserons 4 opérateurs différents :

- *1-flip*, opérateur flippant 1 bit sélectionné aléatoirement parmi les l ;
- *3-flips*, opérateur flippant 3 bits sélectionnés aléatoirement parmi les l ;
- *5-flips*, opérateur flippant 5 bits sélectionnés aléatoirement parmi les l ;
- *1/l-flip*, opérateur flippant chacun des l bits avec une probabilité de $1/l$.

L'algorithme proposé peut donc être vu comme un algorithme de descente avec sélection du premier mouvement améliorant. Son originalité est que l'opérateur de recherche

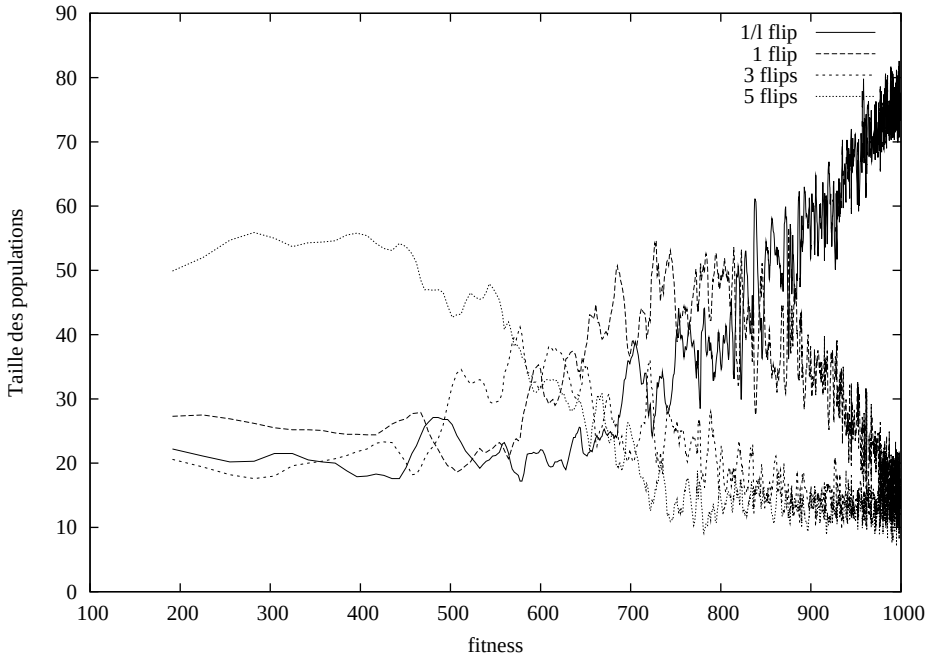


FIGURE 3 – Evolution de la taille de la population des iles en fonction de la fitness globale

locale appliqué à un individu change en fonction de sa fitness. La figure 3 montre l'évolution de la taille des populations des iles correspondant aux opérateurs en fonction de la valeur de la meilleure fitness obtenue depuis le début de la recherche.

4.2 Observations

Nous pouvons observer que l'ile associée à l'opérateur *5-flips* attire le plus les individus en début de recherche. Lorsque la fitness atteint 60% de la fitness optimale (1000 dans notre cas), l'opérateur *3-flips* devient le plus efficace et la probabilité de migration vers son ile associée est donc augmentée (ce qui engendre l'accroissement de la population de son ile). Ensuite, très rapidement, l'opérateur *1-flip* est le plus améliorant et enfin, lorsqu'il devient très difficile d'améliorer les individus, l'opérateur *1/l-flip* apparaît être le plus efficace. Toutes ces observations sont cohérentes avec les résultats déjà observés dans des papiers étudiant l'impact des opérateurs de recherche locale pour le problème ONE-MAX [9]. Notre cadre de travail nous a donc permis de mettre en place un algorithme adaptatif de manière très simple. Pour tester d'autres opérateurs, l'ajout d'autres iles peut être envisagé.

5 Conclusion

Ce travail présente un cadre de travail pour utiliser des modèles en iles dynamiques, permettant tout d'abord de généraliser différents schémas topologiques des modèles classiques, mais surtout d'autoréguler les migrations. Cette modélisation sous forme de graphe complet étiqueté permet de définir tout type de topologie, des graphes sans arc (tous les poids sont nuls, c'est le cas des AG parallèles) aux modèles en iles fréquemment utilisés comme les anneaux ou les treillis. Si la régulation dynamique est activée, alors la topologie est modifiée durant la recherche grâce aux récompenses et pénalités calculées en fonction de l'impact des précédentes migrations. Contrairement aux politiques migratoires des modèles en iles traditionnels, où les migrations sont évaluées *a priori* en mesurant la divergence entre les individus (qui est alors davantage une répartition guidée des individus visant à produire une certaine diversité, mais un non-sens pour un algorithme dit *inspiré par la nature*), nous encourageons (resp. dissuadons) les mouvements migratoires qui ont donné lieu dans un passé proche à de bonnes (resp. mauvaises) descendances, en terme de fitness et / ou de diversité. Durant la recherche, cette autorégulation des probabilités de migrations rend le modèle plus ou moins dynamique en terme de quantités de migrations, ce qui favorise soit la diversification, soit l'intensification (ou les deux, à des endroits différents du modèle).

Les expérimentations réalisées sur les problèmes 0/1 KP et MAX-SAT ont montré qu'un tel schéma dynamique, même avec une paramétrisation basique, apporte de bons résultats, en particulier en comparaison avec les modèles en iles statiques classiques de type anneau unidirectionnel.

Un aspect particulièrement intéressant de notre modèle

est qu'en paramétrant différemment chaque ile, il va affecter de manière autonome ses ressources (individus) sur les iles produisant de meilleurs résultats, c'est-à-dire ayant un paramétrage plus adéquat *à un certain moment de la recherche*. Par exemple, si à chaque ile on associe un opérateur de recherche locale différent, alors le modèle en ile dynamique s'apparentera à un algorithme autonome sélectionnant les opérateurs les plus pertinents au cours de la recherche. C'est ce que nous avons montré empiriquement dans la section 4 sur un problème éprouvé (ONE-MAX), où l'algorithme parvient à répartir les individus sur les opérateurs les plus efficaces, et en étant bien plus réactif que d'autres algorithmes autonomes basés sur l'apprentissage. C'est ce dernier point que nous mesurerons plus précisément dans le cadre de futurs travaux.

Remerciement

Nous souhaitons remercier Sébastien Verel pour ses conseils sur le problème ONE-MAX, ainsi que pour son aide et ses commentaires précieux à propos de la mise à jour des poids dans le cadre de la sélection autonome des opérateurs.

Références

[1] Lourdes Araujo, Juan Julián Merelo Guervós, Carlos Cotta, and Francisco Fernández de Vega. Multikulti algorithm : Migrating the most different genotypes in an island model. *CoRR*, abs/0806.2843, 2008.

[2] Lourdes Araujo, Juan Julián Merelo Guervós, Antonio Mora, and Carlos Cotta. Genotypic differences and migration policies in an island model. In *GECCO*, pages 1331–1338, 2009.

[3] Erick Cantú-Paz. Migration policies, selection pressure, and parallel evolutionary algorithms. *Journal of Heuristics*, 7(4) :311–334, 2001.

[4] Vasek Chvátal. Hard knapsack problems. *Operations Research*, 28 :1402–1411, 1980.

[5] Jörg Denzinger and Jordan Kidney. Improving migration by diversity. In *IEEE Congress on Evolutionary Computation (1)*, pages 700–707, 2003.

[6] I. L. M. Ricarte A. Yamakami E. Noda, A. L. V. Coelho and A. A. Freitas. Devising adaptive migration policies for cooperative distributed genetic algorithms. In *Proceedings of the IEEE International Conference on Systems, Man and Cybernetics*, pages 438–443, 2002.

[7] Taisir Eldos. A new migration model for the distributed genetic algorithms. In *Proceedings of the International Conference on Scientific Computing (CSC ?06), Las Vegas, NV*, pages 26–29.

[8] A. Fialho, L. Da Costa, M. Schoenauer, and M. Sebag. Extreme value based adaptive operator selection. In G. Rudolph et al., editor, *Parallel Problem Solving from Nature - PPSN X, 10th International Conference*, volume 5199 of *Lecture Notes in Computer Science*, pages 175–184. Springer, 2008.

[9] Álvaro Fialho, Luís Da Costa, Marc Schoenauer, and Michèle Sebag. Dynamic multi-armed bandits and extreme value-based rewards for adaptive operator selection in evolutionary algorithms. In *LION*, pages 176–190, 2009.

[10] David E. Goldberg. *Genetic algorithms in search, optimization, and machine learning*. Addison-Wesley, Reading, Mass., 1989.

[11] David E. Goldberg and Kalyanmoy Deb. A comparative analysis of selection schemes used in genetic algorithms. In *Foundations of Genetic Algorithms*, pages 69–93. Morgan Kaufmann, 1991.

[12] David E. Goldberg, Kalyanmoy Deb, and James H. Clark. Genetic algorithms, noise, and the sizing of populations. *COMPLEX SYSTEMS*, 6 :333–362, 1991.

[13] W. Gong, Á. Fialho, and Z. Cai. Adaptive strategy selection in differential evolution. In *Genetic and Evolutionary Computation Conference,GECCO*, pages 409–416. ACM, 2010.

[14] Steven Gustafson and Edmund K. Burke. The speciating island model : An alternative parallel evolutionary algorithm. *J. Parallel Distrib. Comput.*, 66(8) :1025–1036, 2006.

[15] Michail G. Lagoudakis. The 0-1 knapsack problem – an introductory survey. Technical report, University of Southwestern Louisiana, 1996.

[16] Frédéric Lardeux and Adrien Goëffon. A Dynamic Island-Based Genetic Algorithms Framework. In K. Deb et al., editor, *Simulated Evolution and Learning*, volume 6457 of *Lecture Notes in Computer Science*, pages 156–165. Springer, 2010.

[17] Silvano Martello, David Pisinger, and Paolo Toth. Dynamic programming and strong bounds for the 0-1 knapsack problem. *Manage. Sci.*, 45(3) :414–424, 1999.

[18] Zbigniew Michalewicz. *Genetic algorithms + data structures = evolution programs (3rd ed.)*. Springer-Verlag, London, UK, 1996.

[19] Pablo Moscato. Memetic algorithms : a short introduction. pages 219–234, 1999.

[20] Volker Nannen, Selmar K. Smit, and Agoston E. Eiben. Costs and benefits of tuning parameters of evolutionary algorithms. In *Proceedings of the 10th*

international conference on Parallel Problem Solving from Nature, pages 528–538, Berlin, Heidelberg, 2008. Springer-Verlag.

[21] J. N. Richter. *On Mutation and Crossover in the Theory of Evoutionary Algorithms*. PhD thesis, Montana State University, 2010.

[22] Marek Rucinski, Dario Izzo, and Francesco Biscani. On the impact of the migration topology on the island model. *CoRR*, abs/1004.4541, 2010.

[23] Zbigniew Skolicki. *Linkage in Island Models*, chapter Studies in Computational Intelligence, pages 41–60. Springer Berlin / Heidelberg, 2008.

[24] Zbigniew Skolicki and Kenneth A. De Jong. The influence of migration sizes and intervals on island models. In *GECCO*, pages 1295–1302, 2005.

[25] William M. Spears. *Evolutionary Algorithms : The Role of Mutation and Recombination*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2000.

[26] Reiko Tanese. Distributed genetic algorithms. In *Proceedings of the third international conference on Genetic algorithms*, pages 434–439, San Francisco, CA, USA, 1989. Morgan Kaufmann Publishers Inc.

[27] Rasmus K. Ursem. Diversity-guided evolutionary algorithms. In Juan J. Merelo Guervos, Panagiotis Adamidis, Hans-Georg Beyer, Jose Luis Fernandez-Villacanas Martin, and Hans-Paul Schwefel, editors, *Parallel Problem Solving from Nature - PPSN VII, 7th International Conference, Granada, Spain, September 7-11, 2002, Proceedings*, volume 2439 of *Lecture Notes in Computer Science*, pages 462–474. Springer, 2002.

[28] Darrell Whitley, Soraya Rana, and Robert B. Heckendorn. The island model genetic algorithm : On separability, population size and convergence. *Journal of Computing and Information Technology*, 7 :33–47, 1998.

[29] D.H. Wolpert and W.G. Macready. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1(1) :67–82, April 1997.

Un algorithme de résolution d'instances CSP s'appuyant sur les variables FAC

Éric Grégoire **Jean-Marie Lagniez** **Bertrand Mazure**

Université Lille-Nord de France
CRIL - CNRS UMR 8188
Artois, F-62307 Lens

{gregoire,lagniez,mazure}@cril.fr

Résumé

La contribution de ce papier est double. D'une part, il introduit le concept de variable FAC dans le cadre du problème de satisfaction de contraintes (CSP). Ces variables FAC, découvertes à l'aide de la recherche locale, permettent d'augmenter l'efficacité des approches de type MAC. D'autre part, une nouvelle combinaison entre la recherche locale, l'arc-consistance généralisé et un solveur de type MAC permettant d'améliorer la synergie de ces trois approches est proposée. En créant des flux multidirectionnels d'informations entre les différents composants de recherche, cette technique s'avère plus compétitive que les techniques usuelles sur la plupart des classes d'instances.

Abstract

The contribution of this paper is twofold. On the one hand, it introduces a concept of FAC variables in discrete Constraint Satisfaction Problems (CSPs). FAC variables can be discovered by local search techniques and powerfully exploited by MAC-based methods. On the other hand, a novel synergetic combination schema between local search paradigms, generalized arc-consistency and MAC-based algorithms is presented. By orchestrating a multiple-way flow of information between these various fully integrated search components, it proves more competitive than the usual techniques on most classes of instances.

1 Introduction

Le formalisme CSP (*Constraint Satisfaction Problem*) constitue un cadre puissant pour la représentation et la résolution efficace de nombreux problèmes. En particulier, de nombreux problèmes académiques ou réels peuvent être formulés dans ce cadre. À l'heure actuelle, les CSP sont généralement résolus par différentes version d'algorithmes de recherche systématique de type « backtrack ». Une autre possibilité, pour les résoudre, est d'utiliser une méthode de recherche locale (RL) qui est généralement basée sur le concept *min-conflicts* introduit et développé par Minton *et al.* [22]. Bien qu'étant incomplets, c'est-à-dire qu'ils ne fournissent aucune garantie de trouver une solution ou de prouver l'incohérence du problème, ces algorithmes sont souvent efficaces pour trouver une solution. Au vu de son efficacité, elle fut pendant longtemps considérée comme l'approche la plus puissante pour la résolution du problème CSP (et de son homologue SAT) [11, 25, 4].

Cependant, à part dans certains cas spécifiques (*e.g.* [8], [14]), l'approche principalement utilisée pour la résolution de CSP est de type recherche en profondeur d'abord avec retour arrière, et n'inclut généralement pas la recherche locale comme composante principale (*e.g.* Abscon [21, 17], Choco [27], Mistral [13], Sugar [26], etc.). Une des raisons est que la RL n'est pas une méthode complète et n'est pas capable de prouver l'insatisfiabilité d'une formule. De plus, la RL n'utilise aucune des techniques de filtrage (*e.g.* arc-consistance) habituellement employées dans le cadre des approches complètes et qui permettent d'éviter l'exploration d'espaces de recherche non pertinents. Enfin, il est parfois (mais à tort) estimé que la RL devrait simplement être dédiée à la résolution de problèmes où les solutions sont densément réparties dans l'espace de recherche, justifiant ainsi la possibilité d'un certain aspect aléatoire de l'instance considérée.

Ce papier montre que les techniques complètes et la RL peuvent s'avérer complémentaires pour permettre de ré-

soudre le problème CSP. Plus précisément, il présente une combinaison synergique de la recherche locale et d'éléments de techniques complètes, qui souvent surpassent les approches complètes usuelles. Cette méthode n'est pas seulement complète : elle est aussi robuste dans le sens où elle peut aussi bien résoudre les instances satisfiables et insatisfiables, structurées ou aléatoires. En effet notre étude expérimentale montre qu'elle résout globalement plus d'instances que les techniques utilisées couramment.

Une des clés de notre approche est que la recherche locale permet d'extraire les informations nécessaires afin de guider autant que possible la recherche vers les sous-parties difficiles du problème CSP considéré. Bien que cette idée ait déjà été exploitée dans le domaine de SAT [19], elle est ici raffinée grâce au concept original de variables FAC. Ces dernières, dans le cadre CSP, sont des variables qui apparaissent dans toutes les contraintes violées pour une certaine affectation de l'ensemble des variables. Ces variables apparaissent donc dans au moins une contrainte par noyau minimalement incohérent (aussi appelé MUC pour *Minimal Unsatisfiable Core*) d'un CSP. De manière intéressante, la RL permet souvent de détecter efficacement des variables FAC, et permet à une approche complète de type MAC de se focaliser en priorité sur ces variables. De cette manière, les résultats sont significativement améliorés sur de nombreuses instances. De même, des heuristiques puissantes (en particulier $dom/wdeg$ [3]) développées dans le cadre des approches complètes utilisées en CSP peuvent jouer un rôle essentiel dans le processus de RL. Ainsi la méthode proposée, appelée FAC-SOLVER, correspond à la combinaison élaborée de la RL et d'étapes issues de méthodes complètes. Ceci est permis par des échanges réciproques d'informations entre les différentes composantes mises en jeu.

Ce papier est organisé de la manière suivante. Dans la prochaine section les définitions, notations et notions préliminaires sont introduites. Ensuite, le concept de variables FAC est présenté. Dans la section 4, l'architecture de notre approche hybride est détaillée. Avant de conclure et de donner quelques perspectives, l'ensemble des expérimentations conduites est discuté.

2 Définitions, notations et notions préliminaires

Un CSP ou réseau de contraintes $\mathcal{CN}$ est un couple $\langle \mathcal{X}, \mathcal{C} \rangle$ où $\mathcal{X}$ est un ensemble fini de n variables tel qu'à chaque variable X de $\mathcal{X}$ est associé un ensemble fini $dom(X)$ de valeurs. $\mathcal{C}$ est un ensemble fini de m contraintes tel qu'à chaque contrainte C est associée une relation $rel(C)$ indiquant l'ensemble des couples de valeurs autorisés pour les variables $var(C) \subseteq \mathcal{X}$ impliquées par la contrainte C. Une interprétation $\mathcal{I}$ du $\mathcal{CN}$ associe une valeur $\mathcal{I}(X) \in dom(X)$ pour tout $X \in \mathcal{X}$.

Dans la suite, on notera $false(\mathcal{X}, \mathcal{C}, \mathcal{I})$ l'ensemble des variables appartenant à au moins une contrainte violée par l'interprétation $\mathcal{I}$. $\langle \mathcal{X}, \mathcal{C} \rangle|_{X=v}$ est le CSP obtenu lorsque le domaine de la variable X est réduit au singleton $\{v\}$ tandis que $\langle \mathcal{X}, \mathcal{C} \rangle|_{X \neq v}$ est le CSP obtenu en supprimant la valeur v de $dom(X)$. Une interprétation $\mathcal{I}$ se trouve dans un minimum local pour un $\mathcal{CN}$ lorsque $\mathcal{I}$ viole au moins une contrainte du réseau et que quelle que soit l'interprétation voisine $\mathcal{I}'$, c'est-à-dire où une seule valeur est modifiée par rapport à $\mathcal{I}$, le nombre de contraintes violées par $\mathcal{I}'$ ne diminue pas. Une interprétation $\mathcal{I}'$ est voisine de $\mathcal{I}$ si $\exists! X \in \mathcal{X}$ tel que $\mathcal{I}(X) \neq \mathcal{I}'(X)$.

Le problème de satisfaction de contraintes consiste à vérifier si un $\mathcal{CN}$ admet au moins une affectation de ses variables qui satisfait l'ensemble des contraintes du réseau.

L'un des algorithmes de recherche systématique parmi les plus populaires pour résoudre un CSP est MAC (*Maintaining Arc Consistency*) [23]. Cette technique effectue une recherche en profondeur d'abord avec retours-arrière. De plus, à chaque point de choix une consistance locale (AC pour Arc Cohérence) est maintenue. Celle-ci consiste à filtrer les valeurs interdites [18, 2, 16]). Dans la suite de cet article, nous n'imposons aucune limitation à l'arité des contraintes.

3 Les variables FAC

Une des clés de l'efficacité de notre approche est liée au concept de variables FAC présenté ici.

Définition 1. *Soient un réseau de contraintes $\mathcal{CN}$ et une interprétation complète $\mathcal{I}$. Une variable* FAC *est une variable apparaissant dans toutes les contraintes du $\mathcal{CN}$ violées par $\mathcal{I}$.*

Ce concept peut-être rapproché de la notion de *boundary point* introduite par Goldberg dans le cadre du problème SAT [9]. En effet, dans le cadre de SAT, une variable x est considérée comme *boundary* s'il existe une interprétation complète $\mathcal{I}$ construite sur l'ensemble des variables de la formule telle que x appartient à toutes les contraintes falsifiées par $\mathcal{I}$ (ce qui est typiquement la définition d'une variable FAC). Ici, nous avons choisi de ne pas nommer nos variables *boundary* en raison de la nuance suivante : dans le cadre du problème SAT, lorsqu'une variable x est détectée *boundary* pour l'interprétation $\mathcal{I}$, il est possible de satisfaire l'ensemble des contraintes falsifiées par $\mathcal{I}$ en inversant la valeur de vérité de la variable x. Cette propriété permet d'établir qu'une variable *boundary* est à la frontière entre la satisfiabilité et l'insatisfiabilité de la formule. Dans le cadre de CSP, lorsqu'une variable est détectée comme FAC, il n'est pas certain qu'échanger la valeur de cette variable satisfasse la contrainte. Ainsi, la notion de frontière mise en avant par Goldberg ne peut être appliquée ici. C'est pour cela que nous avons choisi

d'utiliser des termes différents.

Les propriétés suivantes peuvent aider à comprendre le rôle potentiel des variables FAC pour la résolution du problème CSP.

Propriété 1. *Soit un réseau de contraintes $\mathcal{CN}$ insatisfiable, si X est une variable* FAC *alors X apparaît dans au moins une contrainte de chaque* MUC *du $\mathcal{CN}$.*

Démonstration. Quelle que soit l'interprétation complète $\mathcal{I}$, au moins une contrainte de chaque MUC est violée par $\mathcal{I}$ (par définition des MUC). Ainsi, si la variable X a été détectée FAC à l'aide de l'interprétation $\mathcal{I}$, alors par définition d'une variable FAC, X apparaît dans toutes les contraintes violées par $\mathcal{I}$. Donc X appartient à au moins une contrainte de chaque MUC. □

Puisqu'elles apparaissent dans tous les MUC, les variables FAC peuvent être considérées comme très intéressantes pour conduire à l'établissement de la preuve de l'incohérence d'un réseau. Cependant établir si une variable est FAC est calculatoirement difficile. En effet, le problème qui consiste à vérifier si une contrainte donnée appartient à au moins un MUC est Σ_2^p [7]. De plus, un CSP peut posséder un nombre exponentiel de MUC. Ce qui implique que, contrôler si une variable est FAC en calculant l'ensemble des MUC du problème est irréalisable dans le pire des cas. Pour pallier ce problème, nous proposons d'utiliser la recherche locale afin de détecter des variables FAC. Pour cela, une méthode de recherche locale classique est lancée. Lorsqu'un minimum local est atteint, l'ensemble des contraintes violées par cette interprétation est parcouru à la recherche de variables FAC. Le fait d'effectuer cette recherche lorsqu'un minimum local est atteint n'est pas fortuit. En effet, il semble raisonnable de ne pas ralentir les performances du solveur en vérifiant chacune des interprétations parcourues par la recherche locale.

Propriété 2. *Un réseau de contraintes $\mathcal{CN}$ qui contient au moins deux* MUC *disjoints (n'ayant aucune variable commune) ne possède pas de variable* FAC.

Démonstration. Soient $\mathcal{CN} = \langle \mathcal{X}, \mathcal{C} \rangle$ un réseau de contraintes, $\mathcal{M}_1 = \langle \mathcal{X}_1, \mathcal{C}_1 \rangle$ et $\mathcal{M}_2 = \langle \mathcal{X}_2, \mathcal{C}_2 \rangle$ deux MUC du $\mathcal{CN}$ tel que $(\mathcal{X}_1 \cap \mathcal{X}_2) = \emptyset$. Raisonnons par l'absurde, supposons qu'il existe $X \in \mathcal{X}$ une variable FAC pour une certaine interprétation $\mathcal{I}$. Par définition des MUC, il existe $C_1 \in \mathcal{C}_1$ et $C_2 \in \mathcal{C}_2$ deux contraintes falsifiées par $\mathcal{I}$. Puisque X est une variable FAC pour $\mathcal{I}$, X appartient à toutes les contraintes falsifiées par $\mathcal{I}$. Donc $X \in C_1$ et $X \in C_2$, ce qui est absurde puisque $(\mathcal{X}_1 \cap \mathcal{X}_2) = \emptyset$. □

La propriété 2 met en avant le fait qu'il n'est pas toujours possible de trouver des variables FAC. Cependant cette situation est rare, puisque dans la plupart des cas les instances insatisfiables ne possèdent pas (ou très rarement) de noyaux inconsistants disjoints, synonyme de deux sources différentes d'incohérence.

Finalement, nous pouvons noter que, dans le cadre d'instances satisfiables, il est tout de même possible de détecter des variables FAC. Ces variables peuvent jouer un rôle important dans la recherche d'une solution en mettant en exergue les parties difficiles du problème. De plus, lors d'un processus de recherche classique (MAC), l'affectation d'une variable peut conduire à obtenir un CSP inconsistant. La détection de variables FAC peut, dans ce contexte, aider à réfuter plus rapidement cette mauvaise décision.

Dans le cadre du problème SAT, l'utilisation de la recherche locale, en prétraitement ou pendant la recherche, a déjà montré son efficacité. Par exemple, Mazure *et al.* [19] ont montré de manière expérimentale qu'effectuer une recherche locale en prétraitement permettait d'obtenir des informations intéressantes concernant les parties difficiles du problème (en particulier les clauses appartenant au noyau minimalement inconsistant). Plus récemment, Audemard *et al.* [1] ont utilisé le concept de *boundary* (proche du concept de variables FAC) et ont montré que l'approche résultante améliorait grandement les performances d'un solveur de type CDCL. L'utilisation du concept de variables FAC et de la recherche locale dans le cadre d'une approche hybride semble donc être très prometteuse. L'approche FAC-SOLVER décrite dans la section suivante tente d'implanter et de valider cela de manière expérimentale en prenant en compte un large panel d'instances.

4 L'approche FAC-SOLVER

Le solveur FAC-SOLVER intègre trois types d'approches en son sein : une recherche locale, un solveur de type MAC et un solveur hybride qui est une combinaison d'un solveur de recherche locale et du processus de filtrage GAC. Ces composantes interagissent ensemble de plusieurs manières (pondération, détection de FAC...) et partagent l'ensemble des informations obtenues au cours de recherche (variables FAC, valeurs supprimées au niveau 0). L'automate décrit dans la Figure 1 donne le schéma général de notre approche. Pour commencer, le processus appelle la recherche locale avec le problème initial. Dans le cas où la recherche locale échoue à trouver une solution dans le temps qui lui a été imparti (contrôlé par $slsProgress$), la partie hybride (RL+GAC) prend la main. Cette partie consiste à rendre taboues les variables problématiques pour la recherche locale en les fixant à l'aide d'un processus complet (affectation et filtrage). Enfin, notre approche passe la main à la partie MAC lorsqu'elle n'a pas été capable de fournir un résultat avant d'avoir atteint un nombre de conflits fixé à l'avance. Cette dernière partie est simplement un solveur MAC classique avec une heuristique de choix de variables basée sur la notion de variable FAC présentée précédem-

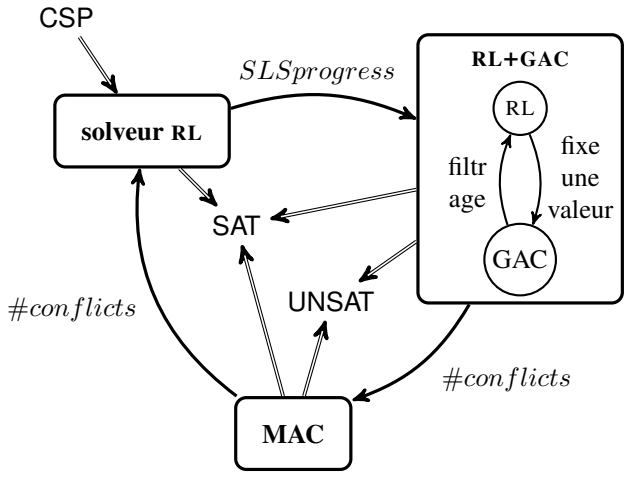


FIGURE 1 – Interactions entre les différentes composantes de FAC-SOLVER

ment. Comme pour la partie RL+GAC, la partie MAC redonne finalement la main à la recherche locale (nouveau cycle) lorsqu'un certain nombre de conflits a été atteint.

4.1 FAC-SOLVER

L'algorithme 1 décrit le solveur FAC-SOLVER. En premier lieu, l'ensemble des variables utiles aux différentes composantes de notre solveur est initialisé : le nombre de conflits contrôlant le redémarrage associé aux parties RL+GAC et MAC du solveur est initialisé à 10, l'ensemble contenant les variables FAC detectées est affecté à vide et une interprétation complète des variables est générée de manière aléatoire (ligne 2-6). Cette interprétation sera utilisée par la recherche locale. Ensuite, un appel à la procédure GAC est effectué afin d'assurer l'arc-consistance ou de montrer que le problème est directement incohérent (lignes 4-5). Une fois ce préambule terminé, tant qu'une solution n'a pas été trouvée ou que l'absence de solution n'a pas été établie, le solveur effectue de manière séquentielle les trois composantes RL, RL+GAC et MAC.

Avant de détailler chacune des composantes, nous décrivons, de manière intuitive, leurs interactions et apports respectifs. Pour la RL, en plus d'essayer de trouver une interprétation satisfaisant l'ensemble des contraintes du CSP, cette composante permet de détecter et collecter des variables FAC. Concernant la partie RL+GAC, elle permet d'aider la recherche locale (en vérifiant que certaines de ces affectations sont bien arc-cohérentes) et permet pondérer les contraintes les plus souvent falsifiées durant la RL (choix des variables limité au *scope* des contraintes falsifiées). Quant à la composante MAC, elle tente de résoudre le problème en intégrant l'ensemble des informations recueillies par les deux autres composantes (FAC et ajustement du poids des contraintes).

Algorithm 1: FAC-solver

Data: Un CSP $\mathcal{CN} = \langle \mathcal{X}, \mathcal{C} \rangle$
Result: $true$ si le CSP est satisfiable, $false$ sinon

1 $result \longleftarrow unknown$;
2 $maxConf \longleftarrow 10$;
3 $\mathcal{S}_e \longleftarrow \emptyset$; `//ensemble de var FAC`
4 $\mathcal{A} \longleftarrow$ un assignement de $\mathcal{X}$ choisi aléatoirement ;
5 `GAC()` ;
6 **if** $\exists X \in \mathcal{X}$ *s.t.* $dom(X) = \emptyset$ **then return** $false$;
7 **while** *(*$result = unkown$*)* **do**
8 initialiser la variable $slsProgress$;
9 `RL(`$\langle \mathcal{X}, \mathcal{C} \rangle$`)` ;
10 **if** *(*$result \neq unkown$*)* **then return** $true$;
11 `Hybrid(`$\langle \mathcal{X}, \mathcal{C} \rangle$`)` ;
12 **if** *(*$result \neq unkown$*)* **then return** $result$;
13 `MAC(`$\langle \mathcal{X}, \mathcal{C} \rangle$`)` ;
14 **if** *(*$result \neq unkown$*)* **then return** $result$;
15 $maxConf \longleftarrow maxConf \times 1.5$;
16 `Backjump(0); //redémarrage/nouveau cycle`

Il important de noter que, puisque $maxConf$ est augmenté de manière géométrique (ligne 16), la composante MAC donnera un résultat lorsque la borne ($maxConf$) deviendra plus grande que le nombre de conflits nécessaires pour résoudre le CSP. Ainsi, la complétude de notre méthode est garantie.

4.2 La composante recherche locale : RL

La procédure 2 présente de manière succincte la composante représentant la partie recherche locale de notre solveur. Cette procédure est basée sur l'approche *walksat* [24] utilisée habituellement dans le cadre de SAT. Elle utilise l'heuristique *novelty* comme critère d'échappement [20] aux minima locaux. Cette méthode identifie également des variables FAC à chaque minimum local. La variable contrôlant l'avancement de la recherche est $slsProgress$, elle est augmentée dans deux situations : quand une nouvelle valeur de maxCSP (nombre de contraintes falsifiées minimum jamais trouvé) est atteinte (ligne 13) et lorsqu'une nouvelle variable FAC est découverte (ligne 4). Elle est décrémentée dans le cas où un minimum local est atteint et qu'aucune variable FAC n'a été trouvée (ligne 6). Cette variable est initialisée à 10000 si le CSP est binaire et 1000 sinon. Cette manière d'estimer le progrès de la RL est inspirée des travaux introduit par Hoos *et al.* [12] concernant le réglage des différents paramètres utilisés en RL dans le cadre de SAT. Ce paramètre est fondamental pour l'efficacité de la l'hybridation et permet d'évaluer dynamiquement la progression de la méthode de recherche locale dans son exploration stochastique de l'espace de recherche. Lorsque la recherche locale échoue à trouver une solution du CSP et qu'elle se trouve « engluée », le test $slsProgress < 0$ (ligne 7) permet de passer la main à

l'algorithme hybride qui va utiliser cette situation d'échec comme point de départ.

Procedure 2 : RL($\langle \mathcal{X}, \mathcal{C} \rangle$)

```
1  while ∃C ∈ C telle que C est falsifiée par A do
2      if minimum local est atteint then
3          if ∃ variable FAC then
4              Ajouter une nouvelle variable FAC à S_e;
5              slsProgress ⟵ slsProgress + 1000 ;
6          else slsProgress ⟵ slsProgress − 1 ;
7          if slsProgress < 0 then return ;
8          else
9              Changer la valeur dans A d'une variable
               de X en fonction du critère d'échappement
               novelty;
10     else
11         Changer la valeur dans A d'une variable de X
           tel que le nombre de contraintes falsifiées
           diminue;
12     if un nouveau maxCSP est obtenu then
13         slsProgress ⟵ slsProgress + 1000 ;
14 result ⟵ true ;
```

4.3 La composante hybrid : RL+GAC

La procédure 3 décrit la partie RL+GAC de notre solveur. Étant donnée la dernière interprétation $\mathcal{A}$ explorée par RL, cette interprétation ne permettait plus à la RL de progresser (suivant notre critère) et va donc être utilisée par la composante RL+GAC pour fixer une valeur à une variable. Tant qu'un certain nombre de conflits n'a pas été atteint (ligne 3), la procédure sélectionne une variable X appartenant à une contrainte falsifiée par $\mathcal{A}$ (en utilisant l'heuristique $dom/wdeg$ [3]) et tente de fixer une de ses valeurs à l'aide de la procédure FIX (ligne 4-6). La valeur est choisie de telle sorte que ce soit la valeur selon laquelle X est assigné dans $\mathcal{A}$. Une fois cette variable fixée le problème peut être montré incohérent ($result = false$) et la procédure se termine. Sinon, un appel à la procédure RL est effectué. Mais les variables fixées durant cette étape ne pourront pas être modifiées par la recherche locale.

La procédure FIX, qui peut être vue comme une partie d'un solveur MAC, permet de gérer la partie affectation et propagation. Elle consiste à assigner la valeur v à la variable X (ligne 1) et à effectuer l'arc-cohérence généralisée sur le problème résultant (ligne 3). Ensuite, tant que le CSP est conflictuel (*i.e.* il existe une variable dont le domaine est vide) la procédure effectue un *backtrack* permettant de rétablir l'état de chaque variable à l'état $level - 1$, décrémente le niveau, réfute la valeur v_{level} de la variable X_{level} affectée au niveau $level$ et effectue de nouveau GAC (lignes 8-12). Un niveau correspond donc au nombre de valeurs

Procedure 3 : Hybrid($\langle \mathcal{X}, \mathcal{C} \rangle$)

```
1  level ⟵ 0;
2  #conf ⟵ 0 ;
3  while (#conf < maxConf) do
4      X ⟵ choisir une variable X ∈ false(X, C, A) à
       l'aide de l'heuristique dom/wdeg;
5      v ⟵ the value of X in A ;
6      FIX(⟨X, C⟩, X, v) ;
7      if (result = false) then return ;
8      RL(⟨X, C⟩) ;
```

fixées heuristiquement par RL+GAC ou MAC.

Procedure 4 : FIX($\langle \mathcal{X}, \mathcal{C} \rangle$, X, v)

```
1  ⟨X, C⟩ ⟵ ⟨X, C⟩|_{X=v};
2  level ⟵ level + 1;
3  GAC() ;
4  while ∃X' ∈ X s.t. dom(X') = ∅ do
5      if level = 0 then
6          result ⟵ false ;
7          return ;
8      ⟨X, C⟩ ⟵ Backtrack() ;
9      level ⟵ level − 1;
10     #conf ⟵ #conf + 1 ;
11     ⟨X, C⟩ ⟵ ⟨X, C⟩|_{X_level ≠ v_level};
12     GAC() ;
```

4.4 La composante MAC

La procédure suivante décrit la procédure MAC.

La composante MAC commence avec le CSP initial modulo les valeurs filtrées au niveau 0 durant les appels précédents à la procédure FIX faits durant RL+GAC. Cette composante diffère des solveurs MAC classiques par l'heuristique de choix de variables. En effet, tant qu'un conflit n'a pas été atteint le prochain point de choix est choisi parmi l'ensemble des variables FAC. Ensuite, l'heuristique de choix de variables devient $dom/wdeg$ (line 8-9) afin de disperser la recherche sur différentes zones d'incohérence. Cet algorithme n'effectue pas nécessairement une recherche complète puisque si le nombre de conflits $\#conf$ devient supérieur à $\#conf$ avant d'avoir résolu le problème, alors le processus s'arrête et un nouveau cycle est réalisé (RL $\rightarrow$ RL+GAC $\rightarrow$ MAC $\hookleftarrow$). Afin de garantir la complétude de la méthode, le nombre de conflits autorisés est augmenté avant chaque nouveau cycle (ligne 15 de l'algorithme 1).

		NOVELTY			RL+GAC			MAC			FAC-SOLVER		
		SAT	UNS	TOT	SAT	UNS	TOT	SAT	UNS	TOT	SAT	UNS	TOT
2-EXT	ACAD	7	0	7	7	2	9	7	2	9	7	2	9
	PATT	106	0	106	100	38	138	83	38	121	99	39	138
	QRND	24	0	24	24	51	75	24	51	75	24	51	75
	RAND	206	0	206	197	105	302	194	110	304	193	106	299
	REAL	6	0	6	7	0	7	7	0	7	7	0	7
	TOTAL	349	0	349	335	196	531	315	201	516	330	198	528
2-INT	ACAD	38	7	45	37	40	77	37	40	77	38	40	78
	BOOL	0	1	1	0	1	1	0	1	1	0	1	1
	PATT	112	0	112	150	60	210	146	62	208	152	62	214
	REAL	47	74	121	74	102	176	75	103	178	75	103	178
	TOTAL	197	82	279	261	203	464	258	206	464	265	206	471
N-EXT	BOOL	70	1	71	74	75	149	74	70	144	74	74	148
	PATT	6	0	6	30	0	30	29	0	29	30	0	30
	QRND	43	0	43	40	40	80	33	40	73	45	40	85
	RAND	70	0	70	68	32	100	72	34	106	70	34	104
	REAL	41	29	70	45	114	159	47	115	162	47	115	162
	TOTAL	230	30	260	257	261	518	255	259	514	266	263	529
N-INT	ACAD	40	0	40	39	23	62	36	23	59	40	23	63
	BOOL	145	1	146	156	12	168	146	12	158	162	13	175
	PATT	88	5	93	103	19	122	95	20	115	102	18	120
	REAL	85	2	87	152	3	155	150	3	153	152	3	155
	TOTAL	358	8	366	450	57	507	427	58	485	456	57	513
	TOTAL	**1134**	**113**	**1247**	**1293**	**717**	**2010**	**1255**	**724**	**1979**	**1317**	**724**	**2041**

TABLE 1 – Résulats expérimentaux.

Procedure MAC ($\langle \mathcal{X}, \mathcal{C} \rangle$)

```
Backjump(0) ;                    // redémarrage
level ⟵ 0;
#conf ⟵ 0 ;
while (#conf < maxConf) do
    if X = ∅ then
        result ⟵ true ;
        return ;
    if (#conf = 0) and (∃X ∈ S_e ∩ X) then
        X ⟵ choisi une variable dans S_e ;
    else
        X ⟵ choisi une variable selon dom/wdeg ;
    v ⟵ choisi aléatoirement une valeur dans
    dom(X) ;
    FIX(⟨X, C⟩, X, v) ;
    if (result = false) then return ;
```

5 Résultats expérimentaux

Afin de valider de manière expérimentale notre approche, nous avons considéré l'ensemble des instances des deux dernières compétitions CSP [5, 6]. Ces instances sont constituées d'instances binaires et n-aires, aléatoires et industrielles, satisfiables et insatisfiables. Ces instances ont été divisées en quatre catégories : 635 instances codant des contraintes binaires en extension (2-EXT), 696 instances codant des contraintes binaires en intention (2-INT), 704 instances codant des contraintes n-aires en extension (N-EXT) et 716 instances codant des contraintes n-aires en intention (N-INT). Afin d'étudier séparément chacune des composantes, nous avons testé quatre méthodes : Walksat+Novelty qui est notre implémentation de la RL, RL+GAC, MAC et notre approche FAC-SOLVER. L'ensemble des tests a été conduit sur un Intel Xeon 3.2 GHz (2 G RAM) sous Linux 2.6. Nous avons limité le temps à 1200 secondes et la mémoire à 900 Mbytes.

Le tableau 1 récapitule les résultats, en terme de nombres d'instances SAT et UNSAT résolues par les différentes approches. Pour chaque catégorie d'instances le total est reporté. Sur ces lignes de totaux, les résultats du meilleur solveur sont grisés. Le principal constat que l'on peut tirer de ce tableau est que notre approche FAC-SOLVER résout globalement le plus d'instances (que ce soit SAT ou UNSAT), et qu'elle est la meilleure sur trois des quatre catégories d'instances considérées. Pour le dernier type d'instances (binaires en extension), nous pouvons voir que les meilleurs solveurs sont différents pour chaque catégorie (SAT, UNS(AT) et TOT(AL)) et que le nombre d'instances résolues par FAC-SOLVER est toujours très proche

du meilleur résultat obtenu.

La Figure 2, représentant cinq nuages de point, permet de comparer plus finement les résultats obtenus par FAC-SOLVER comparativement aux autres approches. Pour chaque couple possible, le nuage de points résultant permet de corréler le temps mis par chacune des deux méthodes pour résoudre une instance donnée. Pour toutes les figures, le solveur FAC-SOLVER est représenté sur l'axe des abscisses tandis que l'approche comparée est reportée sur l'axe des ordonnées. Les résultats, reportés en secondes, sont visualisés à l'aide d'une échelle logarithmique. Ces comparaisons ont été réalisées sur les instances SAT et UNSAT à l'exception de la comparaison avec Walksat+Novelty où seules les instances SAT ont été considérées, la méthode de recherche locale étant incapable de prouver incohérence d'un CSP. Les principales informations pouvant être extraites sont les suivantes :

- Il y a plus d'instances situées sur la ligne Y=1200 que sur la ligne X=1200. Ceci montre, comme cela est reporté dans le tableau 1, que FAC-SOLVER résout plus d'instances que les autres méthodes ;
- À l'exception du solveur MAC sur les instances UNSAT, on peut voir qu'il y a beaucoup de points au dessus de la diagonale, montrant ainsi que FAC-SOLVER est généralement plus efficace que les autres méthodes. De plus, pour les instances UNSAT, on peut voir que la différence entre les différentes paires de solveurs est plus faible que sur les instances SAT (les points sont moins dispersés et plus proches de la diagonale). En ce qui concerne la comparaison de notre approche avec RL+GAC, on peut voir que FAC-SOLVER est globalement plus efficace. Comparativement à l'approche MAC, notre méthode est meilleure sur les instances SAT (apport de la recherche locale) et légèrement moins bonne (en temps) sur les instances UNSAT. Ce décalage de temps sur les instances UNSAT s'explique par le temps utilisé par la recherche locale pour collecter des informations.

Une autre manière de valider la robustesse de notre approche est d'étudier l'impact de l'interprétation initiale choisie pour la partie RL. Pour cela, nous avons sélectionné 96 instances (de manière aléatoire) parmi l'ensemble des instances et pour chacune d'entre elles nous avons lancé notre approche 50 fois avec des interprétations initiales différentes. Les résultats montrent que le choix de l'interprétation ne modifie pas outre mesure les résultats obtenus par notre approche. En effet, lorsqu'une instance a été résolue au moins une fois, elle l'a été dans les 49 autres lancements dans 97 % des cas avec un écart moyen de 2.52 secondes.

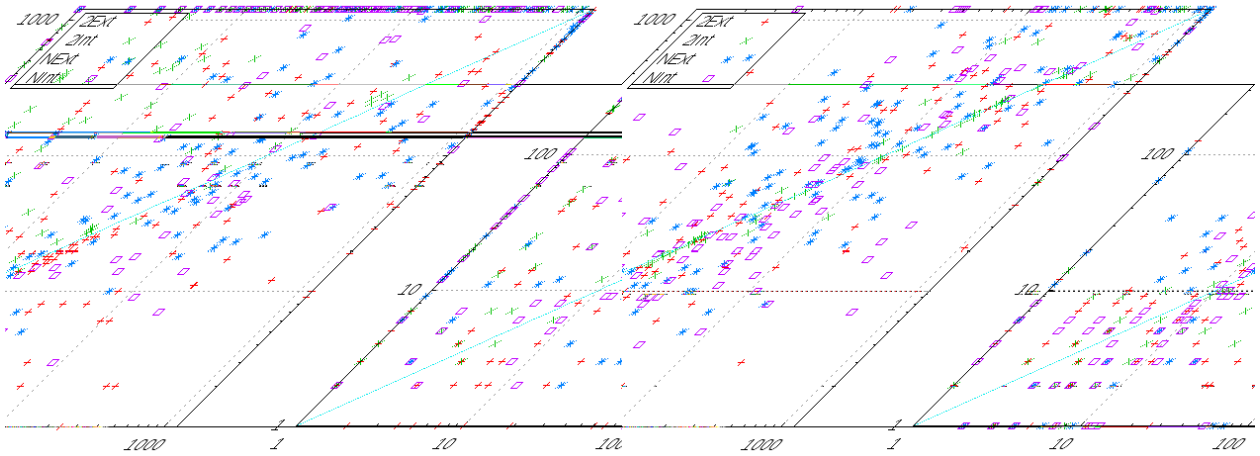
6 Perspectives et Conclusions

Dans ce papier, le concept de variables FAC a été introduit et étudié dans le cadre de la résolution du problème CSP. Le but de cette étude a été de développer un solveur hybride tirant parti de manière efficace des différentes approches présentes en son sein. Les résultats obtenus sur un très large panel d'instances ont permis de montrer que le but que nous nous étions fixé à été atteint.

Une question se pose naturellement : dans quelle mesure chacune des composantes de notre approche prend-elle part à l'amélioration de l'efficacité du solveur ? D'après les expérimentations menées, chacune des composantes permettait de trouver la solution et chacune était nécessaires (variables FAC, RL, méthode hybride impliquant la RL et un processus de filtrage) pour assurer la domination de notre méthode. En particulier, nous avons mesuré que dans 56 % des instances considérées, des variables FAC ont été détectées et ont ainsi permis de jouer un rôle prépondérant dans la résolution du problème (même lorsque celui-ci est satisfiable).

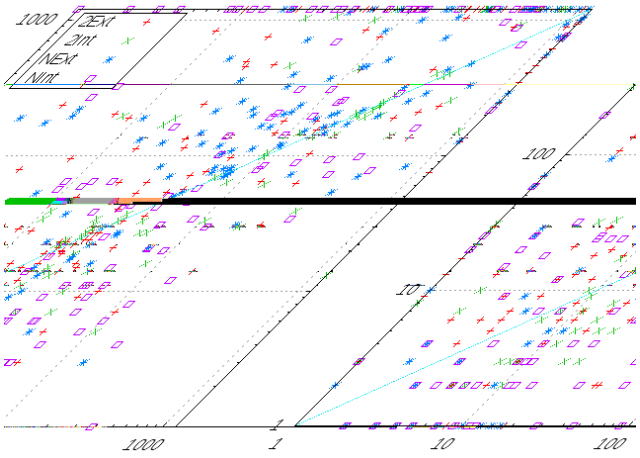
L'algorithme FAC-SOLVER proposé est élémentaire et peut être amélioré par un réglage de ses différents paramètres de plusieurs manières. En particulier, une étude expérimentale plus poussée pourrait permettre d'optimiser les différentes variables de contrôle et les différents facteur d'augmentation, lesquels ont été fixés arbitrairement. De plus, notre implémentation n'inclut pas certaines des techniques de simplification utilisées habituellement dans le cadre de CSP, comme par exemple l'exploitation des symétries ou des contraintes globales. Nous pensons que l'intégration de ces techniques au sein de notre solveur permettrait d'augmenter de manière significative les performances. Il pourrait aussi être intéressant d'explorer une relaxation du concept de variables FAC en prenant en compte (en plus des variables FAC) les variables apparaissant le plus souvent dans les contraintes falsifiées. Cette approche pourrait être utile dans le cas où le CSP considéré ne possède pas de variable FAC.

Finalement, nous pensons que le concept de variables FAC est un bon compromis entre : d'une part le faible coup de calcul nécessaire à la RL pour les détecter, et d'autre part, l'apport théorique concernant l'heuristique de choix de variable d'un solveur MAC afin d'obtenir une preuve de petite taille. Prenant part à l'ensemble des MUC du problème, les variables FAC permettent de se focaliser sur la partie difficile du problème. Cependant, il est facile de trouver des instances insatisfiables où les variables FAC ne prennent conceptuellement pas part à la cause réelle de l'insatisfiabilité, mais apparaissent simplement dans l'ensemble des variables de tous les MUC du problème (alors qu'elles ne fournissent pas véritablement d'information sur la cause du conflit). Raffiner le concept de variable FAC afin de capturer plus finement l'essence de l'insatisfiabilité tout en gardant une heuristique efficace (temps de calcul) constitue un vértiable challenge.

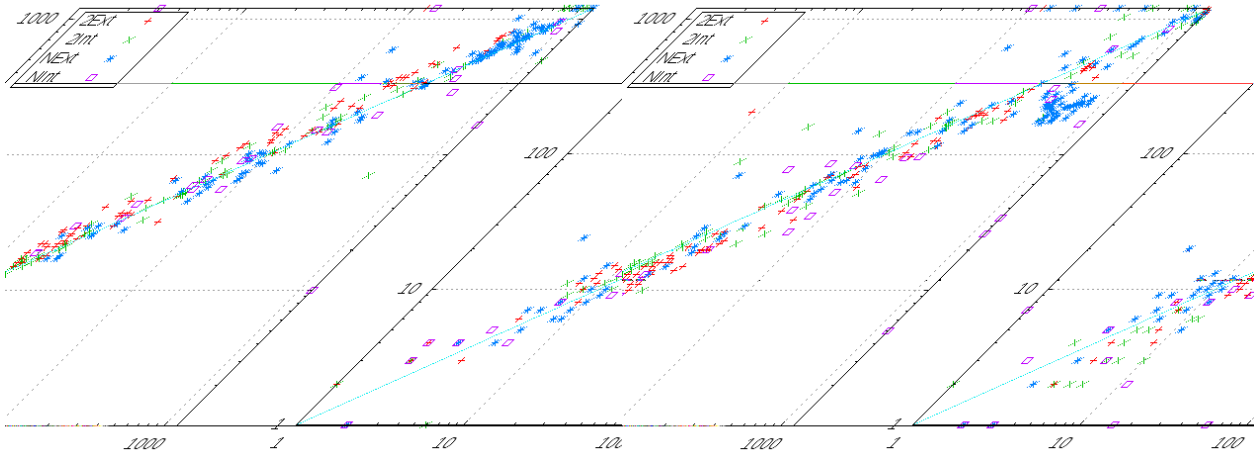


(a) FAC-SOLVER vs. NOVELTY

(b) FAC-SOLVER vs. RL+GAC



(c) FAC-SOLVER vs. MAC



(d) FAC-SOLVER vs. RL+GAC

(e) FAC-SOLVER vs. MAC

FIGURE 2 – Comparaison entre FAC-SOLVER et les autres approches : (a) (b) (c) instances SAT/(d) (e) instances UNSAT.

Références

[1] G. Audemard, J.-M. Lagniez, B. Mazure, and L. Saïs. Boosting local search thanks to cdcl. In *Seventeenth International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR'10)*, pages 474–488, 2010.

[2] 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.

[3] F. Boussemart, F. Hemery, C. Lecoutre, and L. Saïs. Boosting systematic search by weighting constraints. In *Eleventh European Conference on Artificial Intelligence (ECAI'04)*, pages 146–150, 2004.

[4] P. Cheeseman, B. Kanefsky, and W.M. Taylor. Where the really hard problems are. In *Twelfth International Joint Conference on Artificial Intelligence (IJCAI'91)*, pages 331–340, 1991.

[5] Third international csp solver competition, 2008. `http://cpai.ucc.ie/08/`.

[6] Fourth international constraint solver competition, 2009. `http://cpai.ucc.ie/09/`.

[7] T. Eiter and G. Gottlob. On the complexity of propositional knowledge base revision, updates and counterfactuals. *Artificial Intelligence*, 57 :227–270, 1992.

[8] P. Galinier and J.-K. Hao. Tabu search for maximal constraint satisfaction problems. In *Third International Conference on Principles and Practice of Constraint Programming (CP'97)*, pages 196–208, 1997.

[9] E. Goldberg. Boundary points and resolution. In *SAT*, pages 147–160, 2009.

[10] J. Gu. Design efficient local search algorithms. In *Fifth International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems (IEA/AIE'92)*, pages 651–654, 1992.

[11] Holger H. Hoos. An adaptive noise mechanism for walksat. In *AAAI/IAAI*, pages 655–660, 2002.

[12] E. Hébrard. Mistral 1.529, 2006. `http://4c.ucc.ie/~ehebrard/mistral/doxygen/html/main.html`.

[13] N. Jussien and O. Lhomme. Combining constraint programming and local search to design new powerful heuristics. In *Fifth Metaheuristics International Conference (MIC'2003)*, 2003.

[14] C. Lecoutre and F. Hemery. A study of residual supports in arc consistency. In *Twentieth International Joint Conference on Artificial Intelligence (IJCAI'2007)*, pages 125–130, 2007.

[15] C. Lecoutre and S. Tabary. Abscon 112 : towards more robustness. In *Third International Constraint Solver Competition (CSC'08)*, pages 41–48, 2008.

[16] A.K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1) :99–118, 1977.

[17] B. Mazure, L. Saïs, and E. Grégoire. Boosting complete techniques thanks to local search methods. *Annals of Mathematics and Artificial Intelligence*, 22(3-4) :319–331, 1998.

[18] D. McAllester, B. Selman, and H. Kautz. Evidence for invariants in local search. In *Fourteenth National Conference on Artificial Intelligence (AAAI'97)*, pages 321–326, 1997.

[19] S. Merchez, C. Lecoutre, and F. Boussemart. Abscon : A prototype to solve csps with abstraction. In *Seventh International Conference on Principles and Practice of Constraint Programming (CP'01)*, pages 730–744, 2001.

[20] S. Minton, M.D. Johnston, A.B. Philips, and P. Laird. Minimizing conflicts : A heuristic repair method for constraint satisfaction and scheduling problems. *Artificial Intelligence*, 58(1-3) :161–205, 1992.

[21] D. Sabin and E.C. Freuder. Contradicting conventional wisdom in constraint satisfaction. In *Eleventh European Conference on Artificial Intelligence (ECAI'94)*, pages 125–129, 1994.

[22] B. Selman, H. Kautz, and B. Cohen. Noise strategies for improving local search. In *Twelfth National Conference on Artificial Intelligence (AAAI'94)*, pages 337–343, 1994.

[23] B. Selman, H.J. Levesque, and D.G. Mitchell. A new method for solving hard satisfiability problems. In *Tenth National Conference on Artificial Intelligence (AAAI'92)*, pages 440–446, 1992.

[24] N. Tamura, A. Taga, S. Kitagawa, and M. Banbara. Compiling finite linear csp into sat. *Constraints*, 14(2) :254–272, 2009.

[25] Choco Team. Choco : an open source java constraint programming library. Research report 10-02-INFO, Ecole des Mines de Nantes, 2010.

Ajustement dynamique de l'heuristique de polarité dans le cadre d'un solveur SAT parallèle

Long Guo **Jean-Marie Lagniez**

Université Lille-Nord de France
CRIL - CNRS UMR 8188
Artois, F-62307 Lens

{guo,lagniez}@cril.univ-artois.fr

Résumé

Dans cet article, nous proposons une nouvelle heuristique pour la polarité d'affectation d'une variable, dans le cadre d'un solveur SAT parallèle. La polarité selon laquelle le prochain point de choix sera affecté est un processus important des solveurs SAT modernes, en particulier pour les solveurs de type portfolio. En effet, ces solveurs sont souvent basés sur une approche de type coopération/compétition. Dans ce cas, la polarité peut être utilisée pour diriger le solveur dans l'espace de recherche. Nous proposons un critère basé sur le *progress saving* afin d'évaluer si deux solveurs étudient le même espace de recherche. Une fois ce critère établi, nous l'utilisons pour attribuer une heuristique de polarité aux différents solveurs de manière dynamique. Des résultats prometteurs ont été obtenus et montrent l'intérêt de notre approche sur l'affectation de polarité dynamique dans le cadre de solveurs SAT parallèles de type portfolio.

Abstract

In this paper, a new heuristic for polarity selection is proposed. This heuristic is defined for parallel SAT solvers. The selected polarity is an important component of modern SAT solvers, more particulary in portfolio SAT solvers. Indeed, these solvers are often based on the cooperation/competition principle. In this case, the polarity can be used to guide the solver in the search space. A criterion based on the progress saving is proposed in order to evaluate if two solvers study the same search space. Once this criteria defined, a dynamical heuristic polarity is proposed for tuning the different solvers. Experimental results shows that our approach is efficient and increases the capacity of the parallel SAT solver.

1 Introduction

Le problème SAT est un problème central en intelligence artificielle, que ce soit d'un point de vue pratique (planification, vérification formelle, bio-informatique, etc) que d'un point de vue théorique (premier problème à avoir été montré NP-complet). D'un point de vue pratique les performances des solveurs n'ont cessé d'évoluer [13], permettant de résoudre des problème de plus en plus conséquents (des millions de variables et de contraintes). Cette évolution est due, conjointement, à l'avènement des solveurs SAT modernes [16] (*watched literal* , VSIDS, apprentissage) et à l'augmentation de la puissance de calcul (doublant tous les deux ans). Notre époque est témoin d'une « révolution » technologique, qui réside dans l'arrivée des architectures parallèles, permettant d'ouvrir de nouvelles perspectives.

Deux types d'approche pour résoudre SAT en parallèle sont principalement utilisés à l'heure actuelle. D'une part, les approches de type « diviser pour régner » consistent à partager l'arbre de recherche (« *guiding path* ») [14]. D'autre part, les approches de type portfolio [9] mettent les solveurs en concurrence, permettant ainsi de résoudre une formule à l'aide de différentes stratégies. C'est sur ce type d'approche que s'articulent les travaux proposés dans cet article.

Les solveurs SAT modernes sont très sensibles aux réglages de leurs paramètres initiaux [4]. Par exemple, sur certaines instances, changer les paramètres relatifs à la fréquence des redémarrages ou à l'heuristique de choix de polarité peut fortement affecter les performances d'un solveur. Dans ce contexte, une approche de type portfolio permet d'exécuter différentes versions d'un même solveur sur une même instance, profitant ainsi au maximum des ca-

pacités des solveurs SAT modernes.

Afin d'obtenir une approche portfolio efficace, il est nécessaire que les différents paramètres utilisés pour configurer les solveurs tendent à rendre les solveurs complémentaires entre eux. Une des difficultés de ce type d'approche est de paramétrer les différents solveurs de telle manière que deux solveurs n'effectuent pas la même tâche. En effet, dans ce cas, l'une des deux unités de calcul effectue un travail inutile et redondant. Le problème est qu'il est impossible de prévoir, *a priori*, le comportement d'un solveur par rapport à ses paramètres initiaux. Pour pallier à ce problème, nous proposons dans cet article une mesure permettant d'estimer le comportement d'un solveur vis-à-vis d'un autre. Cette mesure est basée sur la notion de *progress saving* [17]. L'intuition est que l'interprétation représentant le *progress saving* peut être vue comme une image de la recherche en cours. De cette manière, deux solveurs peuvent être considérés comme proches s'ils explorent de la même manière l'espace de recherche. Une fois cette mesure définie, celle-ci est utilisée pendant la recherche pour ajuster dynamiquement l'heuristique de choix de polarité. Cet ajustement a pour but d'éloigner deux solveurs considérés comme trop proches.

L'article est organisé de la façon suivante. Après quelques définitions préliminaires (section 2), nous présentons le schéma de base d'un solveur CDCL (*Conflict Driven Clause Learning*) ainsi que deux composantes importantes les caractérisant (heuristique de polarité et stratégie de redémarrage). Une fois ces élément définis, nous présentons le solveur MANYSAT1.1 [9] sur lequel nous avons appliqué notre méthode (section 4). Dans la section 5, nous présentons notre mesure permettant d'estimer si deux solveurs effectuent un travail redondant. Ensuite, une méthode utilisant cette mesure est proposée pour ajuster de manière dynamique l'heuristique de polarité associée à un solveur. Avant de conclure et de donner quelques perspectives, nous donnons des résultats expérimentaux dans la section 6.

2 Définitions et notations

Le problème SAT est le problème de décision qui consiste à vérifier si une formule booléenne Σ sous Forme Normale Conjonctive (CNF), possède ou non une solution. Une formule CNF Σ est un ensemble (interprété comme une conjonction) de *clauses*, où une clause $c_i = (\ell_1 \vee \cdots \vee \ell_{n_i})$ est un ensemble (interprété comme une disjonction) de *littéraux*. Un littéral ℓ est soit une variable positive x ou négative $\neg x$. Le littéral opposé de ℓ est noté $\overline{\ell}$. L'ensemble des variables appartenant à 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 $\mathcal{I} \models \Sigma$. Les notations suivantes seront utilisées par la suite :

- $\#occ(\ell)_\Sigma$ dénote le nombre d'occurrences du littéral ℓ dans la formule Σ (*i.e.* $|\{c \in \Sigma, \ell \in c\}|$). Lorsqu'il n'y a pas d'ambiguïté $\#occ(\ell)_\Sigma$ sera noté $\#occ(\ell)$;
- $\mathcal{C}_i$ dénote une unité de calcul (*i.e.* un solveur).

3 Notions préliminaires

Dans cette section, nous présentons le schéma général d'un solveur CDCL ainsi que deux composantes importantes de ce type de solveur (*i.e.* l'heuristique de choix de polarité et les différentes stratégies de redémarrage).

3.1 Solveur CDCL

Algorithme 1 : Solveur CDCL

Input : une formule CNF Σ
Output : SAT ou UNSAT

```
1  I = ∅ ;                    /* interprétation */
2  dl = 0 ;                   /* niveau de décision */
3  while (true) do
4      γ = propagationUnitaire(Σ,I);
5      if (γ !=null) then
6          β = analyseConflit(Σ,I,γ);
7          bl = calculNiveauBackjumping(β,I);
8          if (bl < 0) then return UNSAT;
9          Σ = Σ ∪ {β};
10         if (restart()) then bl = 0;
11         backjump(Σ,I,bl);
12         dl = bl;
13     else
14         if (I ⊨ Σ) then
15             return SAT;
16         end
17         ℓ = choisirVariableDecision(Σ);
18         dl = dl + 1;
19         I = I ∪ {selectPhase(ℓ)};
20     end
21 end
```

L'algorithme 1 donne le schéma général d'un solveur de type CDCL. Typiquement, un tel solveur peut être assimilé à une séquence de décision et de propagation des littéraux unitaires. Chaque littéral choisi comme littéral de décision (lignes 18-20) est affecté suivant une certaine polarité à un niveau de décision (dl). Si tous les littéraux sont affectés, alors $\mathcal{I}$ est un modèle de Σ (lignes 16-17). À chaque fois qu'un conflit est atteint par propagation (γ est la clause conflit), un *nogood* β est calculé (ligne 8) en

utilisant une méthode d'analyse de conflits donnée, le plus souvent selon le schéma « First UIP » (*« Unique Implication Point »* [18]), et un niveau de *backjump* bl est calculé. À ce moment, il est possible de prouver l'inconsistance de la formule (ligne 10). Si ce n'est pas le cas, on fait un saut arrière et le niveau de décision devient égal au niveau de *backjump* (ligne 13-14). Finalement, certains solveurs CDCL forcent le redémarrage (diverses stratégies sont possibles, voir [11] par exemple) et dans ce cas, on remonte en haut de l'arbre (ligne 12).

3.2 Heuristique de sélection de polarité

L'heuristique de choix de variable est l'une des composantes les plus importantes des solveurs SAT modernes. La plus utilisée à l'heure actuelle est VSIDS (*Variable State Independent Decaying Sum*) [16]. Chaque fois qu'une nouvelle variable x est sélectionnée comme point de choix, il est nécessaire de lui associer une polarité (x ou $\neg x$). Dans le solveur MINISAT [7] une heuristique de phase très simple a été proposée. Celle-ci consiste à toujours affecter la variable à la valeur fausse (*false* : $\neg x$). Jeroslaow et Wang [12], quant à eux, proposent une mesure w, prenant en compte des informations sur le problème (taille et nombre de clauses contenant la variable), pour selectionner la polarité du prochain point de choix (x si $w(x) > w(\neg x)$, $\neg x$ sinon). Une manière simple d'utiliser cette heuristique est de considérer le nombre d'occurrences d'un littéral comme mesure (*occurence* : x si $\#occ(x) > \#occ(\neg x)$, $\neg x$ sinon). Dans [17], Pipatsrisawat et Darwish observent, de manière expérimentale, que dans le cadre d'une approche utilisant le *backtracking* les solveurs effectuaient beaucoup de travail redondant. En effet, lorsqu'un retour-arrière est effectué le travail accompli pour résoudre les sous problèmes traversés avant d'atteindre le conflit est perdu. Pour éviter cela, les auteurs proposent de sauvegarder la dernière polarité obtenue, pendant la recherche, dans une interprétation complète, noté $\mathcal{P}$. Ainsi, lorsqu'une nouvelle décision sera prise, la variable se verra attribuée la même polarité que précédemment (*progress saving* : x si $x \in \mathcal{P}$, $\neg x$ sinon). De cette manière, l'effort pour satisfaire un sous-problème déjà résolu auparavant sera moins important.

3.3 Différentes stratégies de redémarrage

La politique de redémarrage est une composante importante des solveurs SAT modernes [8]. Sommairement, le principe consiste à remonter l'arbre de recherche jusque la racine, tout en conservant les informations obtenues jusqu'ici (pondération des variables, clauses apprises, ...), lorsqu'un certain nombre de conflits a été atteint. Différentes politiques de redémarrage ont été proposées. Dans le cadre de cet article, nous ne présentons que les politiques de redémarrage utiles à la compréhension (pour plus d'information sur le sujet le lecteur pourra consulter [11]). Le nombre de conflits pouvant être rencontrés au $n^{\text{ème}}$ redémarrage (avant d'effectuer un retour-arrière à la racine) peut être calculé à l'aide des fonctions suivantes :

- $geometrical(x, y)$: stratégie dans lequelle on a un redémarrage tous les x conflits. La valeur x est multipliée par y à chaque redémarrage ;
- $luby(x)$: le $i^{\text{ème}}$ terme de la suite de Luby[15] se calcule comme suit : $t_i = 2^{k-1}$ si $i = 2^{k-1}$, sinon $t_i = t_{i-2^{k-1}+1}$. Le nombre de conflits à atteindre avant d'effectuer le $n^{\text{ème}}$ redémarrage est égal à $t_n \times x$;
- $dynamic^{+}(x_1, x_2)$: cette fonction est calculée comme suit : on a $x_i = y_i \times |cos(1 + r_i)|$, avec $i > 2$, $\alpha = 1200$, y_i représente la moyenne de la taille des retours-arrière au redémarrage i, $r_i = \frac{y_{i-1}}{y_i}$ si $y_{i-1} > y_i$ et $r_i = \frac{y_i}{y_{i-1}}$ sinon ;
- $dynamic^{-}(x_1, x_2)$: cette fonction est une variante de $dynamic^{+}$ où $x_i = y_i \times |cos(1 - r_i)|$.

L'heuristique $dynamic$ a été proposée par Jabbour *et al.* et a été implémentée dans le solveur LySAT [10].

4 Solveur parallèle

Il existe principalement deux types d'approches pour résoudre SAT en parallèle. D'une part, les approches de type « diviser pour régner » qui consistent à partager l'arbre de recherche (« *guiding path* ») [14]. D'autre part, les approches de type portfolio [9] où les solveurs sont mis en concurrence permettant ainsi de résoudre une formule à l'aide de différentes stratégies. Dans cette article, seule l'approche portfolio est présentée. Nous détaillons plus particulièrement MANYSAT1.1, qui est le solveur utilisé pour la partie expérimentale de nos travaux. La figure 1 représente de manière schématique le fonctionnement d'un solveur parallèle.

Comme présenté plus haut, MANYSAT1.1 est un solveur parallèle de type portfolio. Dans MANYSAT1.1, chaque coeur représente un solveur séquentiel contenant toutes les stratégies classiques (*i.e.* propagation unitaire, *watched litterals*, VSIDS, analyse de conflits, etc). De plus, lorsqu'un solveur apprend une clause (*i.e.* un conflit est atteint), celle-ci est transférée aux autres solveurs (si sa taille est inférieure à 8). Ce type d'approche est basé sur le principe de compétition/coopération. Compétition, dans le sens où les différents solveurs ont des stratégies différentes. Et coopération, dans le sens où les solveurs se partagent des informations par le biais des clauses apprises. Le tableau 1 détaille les différentes stratégies utilisées dans MANYSAT1.1. En ce qui concerne la stratégie d'apprentissage utilisée par le coeur 3, celle-ci a été proposée dans [2] et est basée sur le principe d'extension du graphe d'implication classique à l'aide d'arcs inverses. Ces arcs inverses, sont obtenus en prenant en compte les clauses satisfaites dans l'analyse du graphe d'implication.

Stratégie	Coeur 1	Coeur 2	Coeur 3	Coeur 4
Redémarrage	$dynamic^{+}$	$dynamic^{-}$	$geometrical(100, 1.5)$	$dynamic^{+}$
Heuristique	VSIDS (2% rand.)	VSIDS (2% rand.)	VSIDS (2% rand.)	VSIDS (3% rand.)
Polarité	*progress saving*	*progress saving*	*false*	*occurence*
Apprentissage	first-UIP	first-UIP	first-UIP étendu	first-UIP
Partage de clauses	taille ≤ 8	taille ≤ 8	taille ≤ 8	taille ≤ 8

TABLE 1 – Stratégie des différents solveurs séquentiels utilisés dans le solveur MANYSAT1.1.

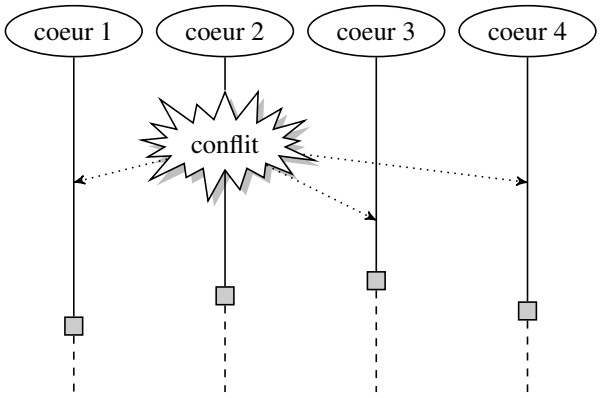


FIGURE 1 – Présentation schématique d'un solveur parallèle de type portfolio. Les lignes continues représentent l'exécution d'un solveur. Les carrés symbolisent les redémarrages tandis que les flèches en pointillés représentent le partage d'information entre les différents solveurs (ici les clauses apprises).

5 Utilisation du *progress saving* pour estimer la distance entre solveurs

Comme précisé précédemment, les solveurs parallèles de types portfolio sont souvent basés sur le principe de compétition/coopération. Ainsi dans le solveur MANYSAT1.1, la phase de coopération peut être assimilée au transfert de clauses apprises. Tandis que la phase de compétition peut être associée aux différentes stratégies choisies (redémarrage, heuristique de polarité, ...) pour chacun des solveurs. Contrairement à la partie coopérative du solveur, la manière dont les solveurs entrent en compétition est choisi de manière statique au début de la recherche. Ce type d'approche ne permet pas d'identifier ni de traiter le cas où deux solveurs effectuent le même travail. En effet, bien que deux solveurs aient des stratégies différentes, il n'est pas sûr qu'ils entrent en compétition. Il peut même arriver des cas où l'opposé se produit, c'est-à-dire que les deux solveurs vont entrer en phase de coopération. Cette coopération se traduit le plus souvent par un travail redondant de la part d'un des deux solveurs.

Pour éviter ce genre de situation, et ainsi préserver un schéma de type compétition/coopération, une méthode permettant d'ajuster dynamiquement l'heuristique de choix de la polarité est proposé. Pour cela, une mesure permettant d'estimer la distance entre deux solveurs est introduite. Afin d'étudier le comportement des solveurs vis-à-vis de notre mesure, nous avons effectué un ensemble d'expérimentations.

5.1 Distance entre deux solveurs

Pour savoir si deux solveurs sont en train d'effectuer le même travail, nous avons choisi d'utiliser le *progress saving* $\mathcal{P}$. En effet, il est possible de considérer l'interprétation complète $\mathcal{P}$ comme une image de la recherche en cours. De cette manière, on peut supposer que deux solveurs sont voisins dans l'espace de recherche lorsqu'ils ont des *progress saving* proches (proche au sens de distance de Hamming). De cette analyse, il est possible de définir une mesure permettant d'estimer la distance entre deux solveurs. Cette mesure sera notée δ.

Définition 1 *Soient Σ une formule* CNF*, $(\mathcal{C}_i, \mathcal{P}_i)$ et $(\mathcal{C}_j, \mathcal{P}_j)$ deux couples de solveurs avec leurs progress savings respectifs, on note :* $\delta(\mathcal{C}_i, \mathcal{C}_j) = 1 - \frac{|\mathcal{P}_i \cap \mathcal{P}_j|}{|\mathcal{V}_\Sigma|}$.

Remarquons que deux solveurs sont proches, d'après notre mesure, si leurs interprétations représentant leurs *progress savings* respectifs ont une petite distance de hamming.

5.2 Évolution de la distance entre différents solveurs

Afin d'étudier l'évolution de la distance entre différents solveurs entre eux, nous avons exécuté le solveur MANYSAT1.1 en récupérant la distance entre les différents solveurs tous les 5000 conflits. Les courbes de la figure 2 retranscrivent, de manière représentative, le comportement des différents solveurs les uns envers les autres. On peut constater que les solveurs se divisent en deux catégories. La première catégorie regroupe les courbes où le coeur 4 apparaît et la seconde catégorie les autres. Concernant les courbes de la première catégorie, on observe que la distance entre le coeur 4 et n'importe quel autre coeur est toujours supérieure au cas où le coeur 4 n'est pas présent dans

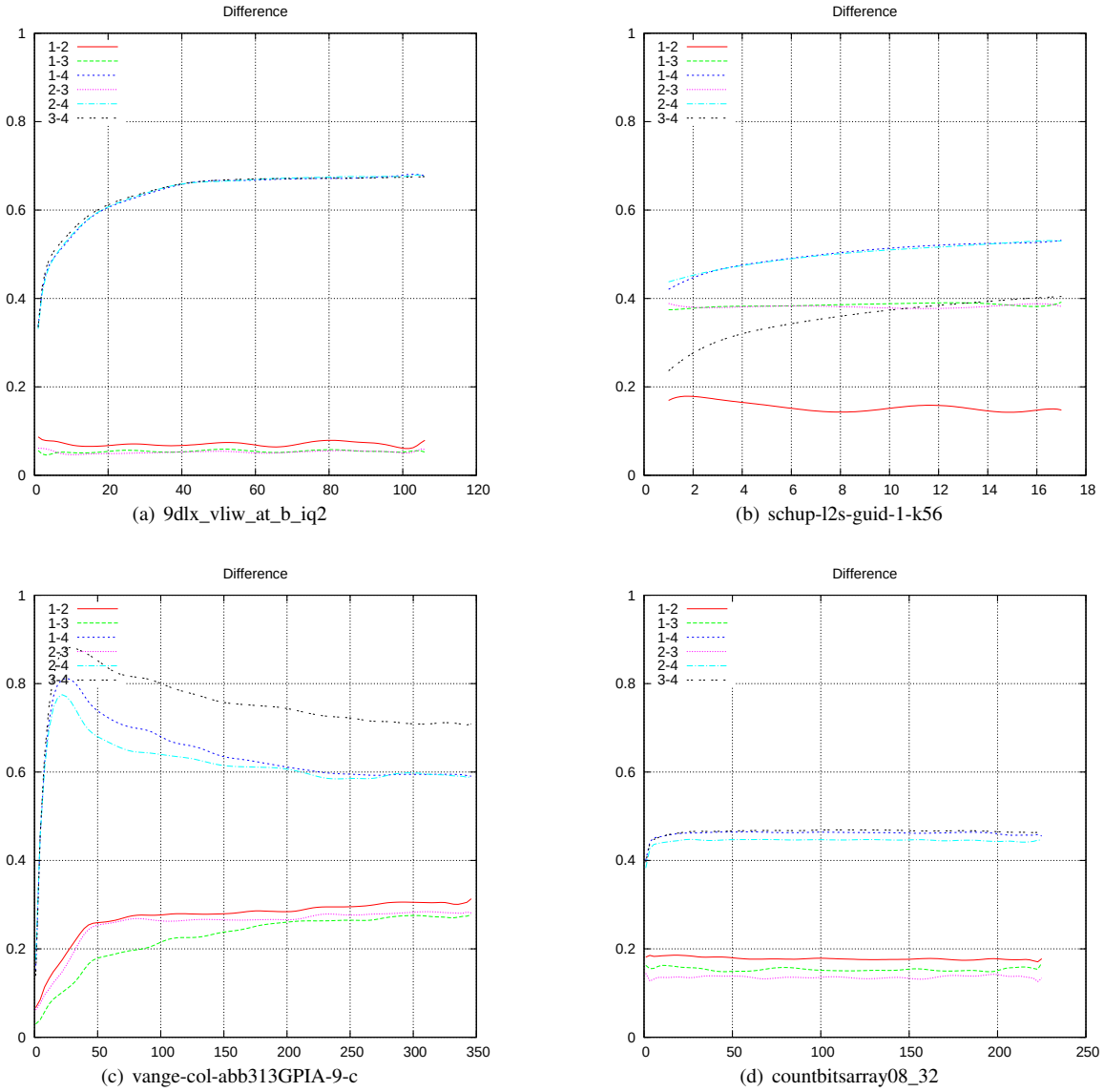


FIGURE 2 – Comparaison deux à deux de la distance entre différents solveurs exécutés en parallèle. Sur l'axe des abscisses est représenté le nombre de conflits atteint (divisé par 5000) tandis que l'axe des ordonnées représente le résultat de notre mesure. Pour une meilleure lisibilité toutes les courbes ont été lissées.

le calcul de la distance. Ceci peut être expliqué par le choix de la stratégie d'affectation de polarité du coeur 4. En effet, son heuristique de choix de polarité est basée sur le nombre d'occurrences d'un littéral dans la formule. Étant donné que les solveurs s'échangent de nombreuses clauses, il y a une forte variation du nombre d'occurrences d'un littéral, ceci a pour conséquence de grandement diversifier l'espace de recherche exploré par le coeur 4. Pour se qui est de la seconde catégorie, on peut voir que les coeurs 1, 2 et 3 sont beaucoup plus proches les uns des autres. Ceci est facilement explicable pour les coeur 1 et 2, étant donné qu'ils utilisent tous deux la même heuristique de choix de polarité. En ce qui concerne le coeur 3, nous pensons que cela est du au choix de la polarité initiale pour le *progress saving* (la phase étant affectée à « faux »).

5.3 Ajustement dynamique de la polarité

Après analyse du comportement des différents solveurs entre eux, nous avons choisi de définir une nouvelle heuristique de polarité. Puisque les solveurs peuvent être très proches dans l'espace de recherche nous avons décidé d'ajouter du bruit pour les éloigner les uns des autres. Pour cela, lorsque deux coeurs sont détectés comme « trop proches », nous choisissons d'inverser la polarité d'un des

deux coeurs. Pour éviter que deux solveurs proches n'inversent en même temps leur polarité, l'ajustement se fait de manière unidirectionnelle, c'est-à-dire qu'un seul des deux solveurs ajuste sa polarité. De plus, afin de ne pas ralentir de manière excessive la vitesse du solveur, cette ajustement ne sera pas fait de manière systématique. La figure 3 décrit le transfert de messages et l'ajustement de la polarité entre deux solveurs. À chaque point de contrôle le coeur b va demander au coeur a s'ils sont proches. Le coeur a lui répond par oui ou par non en fonction du résultat obtenu en mesurant sa distance avec le coeur b à l'aide de notre mesure. Dans le cas où la réponse est négative, le coeur b utilise son heuristique de choix de polarité initial. Dans le cas d'une réponse positive, lorsque le coeur b sera de nouveau amené à choisir la polarité d'un point de choix, il inversera simplement son heuristique de polarité. Par exemple, supposons que le coeur b aie comme heuristique de polarité l'heuristique *false*. Dans ce cas, tant qu'un nouveau point de contrôle n'est pas atteint, le solveur affectera les prochains points de choix à vrai. Une fois le point de contrôle atteint, l'heuristique de polarité peut de nouveau être réajustée. Ce processus est répété tant qu'une solution n'a pas été trouvée ou que le problème n'a pas été réfuté.

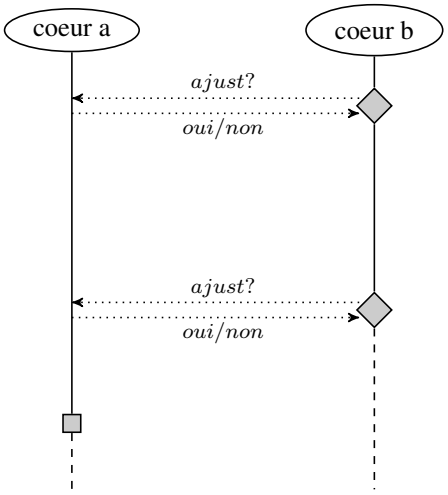


FIGURE 3 – Ajustement dynamique de la polarité d'un solveur vis-à-vis d'un autre.

6 Expérimentations

Afin d'étudier l'impact de notre méthode, les expérimentations ont été divisées en deux parties. Dans la première, nous avons choisi d'isoler l'heuristique de choix de polarité vis-à-vis des autres paramètres du solveur. Pour cela, hormis l'heuristique de choix de polarité, l'ensemble des unités de calcul utilisent des architectures identiques (*i.e.* même politique de redémarrage, même stratégie d'analyse de conflit et même fréquence du choix aléatoire de la prochaine variable à affecter). Dans la seconde partie, nous avons intégré directement notre méthode au solveur MANYSAT1.1.

Avant d'entamer les expérimentations, il nous a fallu définir un schéma d'application pour notre méthode. En effet, dans la section précédente nous ne définissons pas formellement le cadre selon lequel notre ajustement doit être effectué.

Tout d'abord, à la vue des résultats obtenus concernant l'évaluation de la distance entre les solveurs entre eux, notre méthode ne sera pas appliquée sur le coeur 4 (coeur utilisant l'heuristique de polarité *occurence*). Ensuite, afin d'éviter des problèmes de cycle dans le protocole d'ajustement de polarité notre approche ne sera pas non plus appliquée au coeur 1. La figure 4 schématise la manière selon laquelle l'ajustement des coeurs 2 et 3 sera effectué. Nous avons choisi d'ajuster le coeur 2 dans le cas où il est proche ($\delta(\mathcal{C}_1, \mathcal{C}_2) \leq 0.1$) du coeur 1. L'heuristique de polarité du coeur 3 sera ajustée dans le cas où $\delta(\mathcal{C}_1, \mathcal{C}_3) \leq 0.1$ et que $\delta(\mathcal{C}_1, \mathcal{C}_2) > 0.1$. Les contrôles seront effectués tous les 5000 conflits.

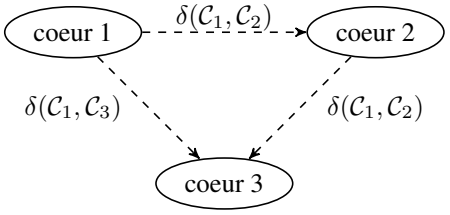


FIGURE 4 – Schéma d'ajustement.

L'ensemble des résultats expérimentaux reportés dans cette section ont été obtenus sur un Quad-core Intel XEON X5550 avec 32Gb de mémoire. Le temps CPU est limité à 900 secondes par unité de calcul. Pour ces expérimentations nous avons utilisé les 292 instances industrielles de la dernière compétition SAT [1]. Toutes les instances sont pré-traitées à l'aide de SATElite [6]. Étant donnée le non déterminisme, lié à la résolution en parallèle du problème, chaque solveur sera lancé trois fois. Les résultats reportés sont selectionnés parmi le meilleur des trois lancements.

6.1 Architecture identique

Comme précisé précédemment, afin d'isoler l'heuristique de polarité des autres composantes des solveurs SAT modernes, nous avons choisi d'utiliser une architecture identique pour l'ensemble des coeurs. En ce qui concerne l'heuristique de polarité nous avons conservé celle implémentée dans la version initiale de MANYSAT1.1 (voir figure 1). Sur les figures 5 et 6, nous pouvons constater que, sur les résultats obtenus sur l'ensemble des instances industrielles de la dernière compétition SAT, de notre technique (74 SAT et 120 UNSAT) permet d'améliorer sensible-

ment les performances du solveurs (72 SAT et 111 UNSAT). Plus particulièrement, le nuage de point 6 nous montre que l'ajout de notre méthode permet toujours de résoudre de manière plus efficace les instances insatisfiables. Concernant les instances satisfiables nous pouvons noter, même si les points sont cette fois-ci plus éparpillés, que notre approche est la meilleure.

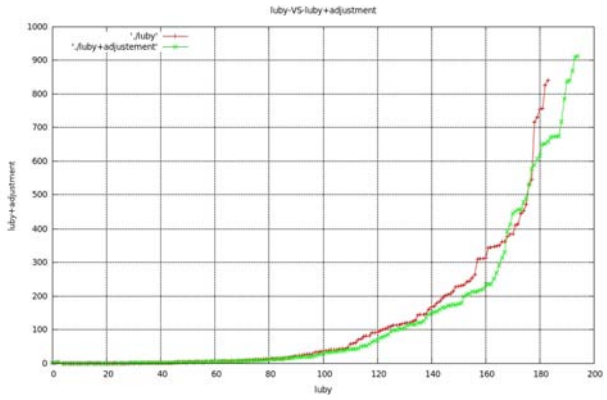


FIGURE 5 – Nombre d'instances résolues en fonction du temps pour le solveur parallèle MANYSAT1.1 intégrant ou pas notre technique d'ajustement de polarité, dans le cas d'une architecture identique pour tous les coeurs.

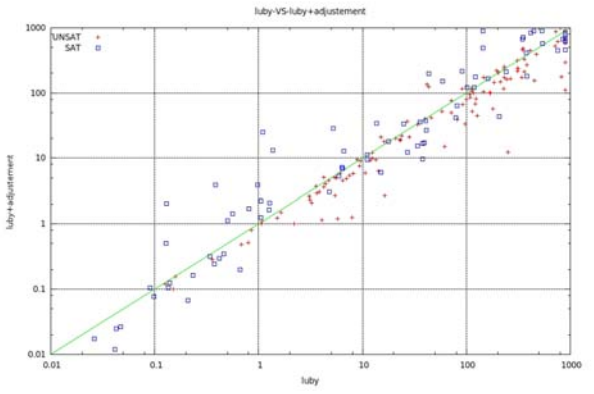


FIGURE 6 – Corrélation entre le temps mis par chacune des deux méthodes (MANYSAT1.1 architecture identique et MANYSAT1.1 architecture + ajustement) pour résoudre une instance donnée.

6.2 Ajustement de la polarité dans MANYSAT1.1

Ici nous évaluons le gain apporté par notre technique d'ajustement de polarité dans le cadre du solveur MANYSAT1.1, lequel résout 73 instances satisfiables et 120 instances insatisfiables. Les résultats obtenus et reportés sur les figures 7 et 8 montrent que l'ajout de notre méthode au sein du solveur SAT parallèle MANYSAT1.1 permet d'améliorer sa compétitivité (81 SAT et 123 UNSAT). Comme dans le cas où les architectures sont identiques, nous pouvons voir que l'ajout de notre méthode permet au solveur de résoudre plus efficacement les instances insatisfiables. En ce qui concerne les instances satisfiable les résultats montrent que notre approche permet de résoudre sensiblement plus d'instances.

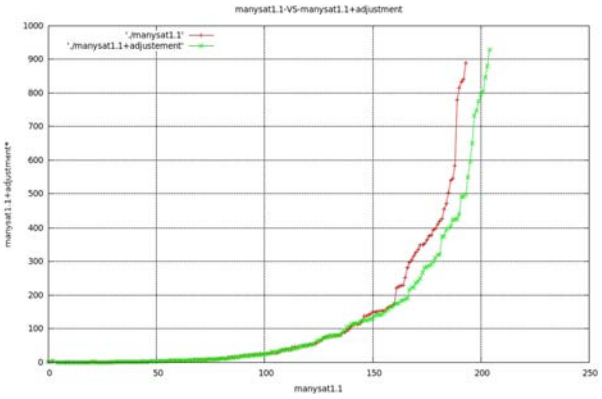


FIGURE 7 – Nombre d'instances résolues en fonction du temps pour le solveur parallèle MANYSAT1.1 intégrant ou pas notre technique d'ajustement de polarité.

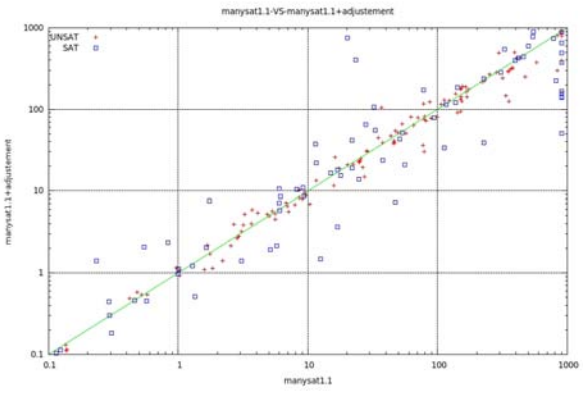


FIGURE 8 – Corrélation entre le temps mis par chacune des deux méthodes (MANYSAT1.1 et MANYSAT1.1 + ajustement) pour résoudre une instance donnée.

7 Conclusion et perspectives

Des travaux sur l'utilisation du *progress saving* pour régler certaines composantes des solveurs SAT modernes ont déjà été menés par le passé. Ainsi, dans [5], une politique de redémarrage basée sur le *progress saving* est proposée. Plus récemment, Audemard *et al.* [3] ont proposé d'utiliser cette notion pour contrôler la base de clauses apprises. À notre tour, en utilisant une approche d'ajustement très simple basé sur le *progress saving*, nous avons pu améliorer les performances d'un solveur SAT de type portfolio.

Cette approche consiste à régler de manière dynamique l'heuristique de choix de polarité. Notre méthode permet d'estimer la distance séparant deux solveurs à l'aide du *progress saving*. L'avantage de cette mesure est de fournir des informations sur les différents coeurs, permettant ainsi d'agir sur l'heuristique de choix de polarité pendant la recherche. Les résultats expérimentaux, menés sur l'ensemble des instances industrielles de la dernière compétition SAT, ont montré que notre méthode améliore sensiblement les performances du solveur MANYSAT1.1 auquel a été greffée notre méthode. Une étude du comportement de notre approche sur les autres familles d'instances est envisagée. De plus, il semble que la fréquence selon laquelle l'ajustement est effectué influe sur les performances. Il semble donc nécessaire d'étudier plus finement ce phénomène et de définir une stratégie dynamique pour régler ce paramètre.

Au lieu de se réduire à inverser l'heuristique de choix de polarité comme nous l'avons fait ici, une autre approche pourrait consister à en changer complètement pendant la phase de recherche. De cette manière un portfolio d'heuristique de polarité serait obtenu. Une tout autre perspective consisterait à utiliser les informations fournies par le *progress saving* pour contrôler les flux de clauses apprises transitant entre les différents coeurs d'un solveur parallèle.

Remerciements

Nous tenons à remercier Elodie Vandenhaute et Gilles Audemard pour les commentaires qu'ils nous ont apportés sur la rédaction de cet article.

Références

[1] « 12th international symposium on theory and applications of satisfiability testing ». May 2009.

[2] Gilles Audemard, Lucas Bordeaux, Youssef Hamadi, Saïd Jabbour, and Lakhdar Sais. A generalized framework for conflict analysis. In *SAT*, pages 21–27, 2008.

[3] Gilles Audemard, Jean Marie Lagniez, Bertrand Mazure, and Lakdhar Saïs. RCL : reduce learnt clauses. SATRACE, solver description, 2010.

[4] A. Biere, M. Heule, H. van Maaren, and T. Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, February 2009.

[5] Armin Biere. Adaptive restart strategies for conflict driven sat solvers. In *SAT*, pages 28–33, 2008.

[6] N. Eén and A. Biere. Effective preprocessing in SAT through variable and clause elimination. In *SAT*, pages 61–75, 2005.

[7] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In *SAT*, pages 502–518, 2003.

[8] Carla P. Gomes, Bart Selman, Ken McAloon, and Carol Tretkoff. Randomization in backtrack search : Exploiting heavy-tailed profiles for solving hard scheduling problems. In *AIPS*, pages 208–213, 1998.

[9] Youssef Hamadi, Saïd Jabbour, and Lakhdar Sais. Manysat : a parallel sat solver. *JSAT*, 6(4) :245–262, 2009.

[10] Youssef Hamadi, Saïd Jabbour, and Lakhdar Saïs. Lysat : Solver description. SATRACE, solver description, 2010.

[11] J. Huang. The effect of restarts on the efficiency of clause learning. In *proceedings of IJCAI*, pages 2318–2323, 2007.

[12] Robert G. Jeroslow and Jinchang Wang. Solving propositional satisfiability problems. *Ann. Math. Artif. Intell.*, 1 :167–187, 1990.

[13] Daniel Le Berre. Sat4j, un moteur libre de raisonnement en logique propositionnelle, dec 2010.

[14] Matthew D. T. Lewis, Tobias Schubert, and Bernd Becker. Multithreaded sat solving. In *ASP-DAC*, pages 926–931, 2007.

[15] Michael Luby, Alistair Sinclair, and David Zuckerman. Optimal speedup of las vegas algorithms. In *ISTCS*, pages 128–133, 1993.

[16] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff : Engineering an efficient SAT solver. In *DAC*, pages 530–535, 2001.

[17] Knot Pipatsrisawat and Adnan Darwiche. A lightweight component caching scheme for satisfiability solvers. In *SAT*, pages 294–299, 2007.

[18] L. Zhang, C. Madigan, M. Moskewicz, and S. Malik. Efficient conflict driven learning in boolean satisfiability solver. In *ICCAD*, pages 279–285, 2001.

Concilier Parallélisme et Déterminisme dans la Résolution de SAT

Youssef Hamadi [1,2] **Said Jabbour** [3] **Cédric Piette**[3] **Lakhdar Saïs**[3]

[1] Microsoft Research
7 J J Thomson Avenue
Cambridge, United Kingdom

[2]LIX École Polytechnique, F91128 Palaiseau, France

[3]Université Lille-Nord de France, Artois, F-62307 Lens
CRIL, F-62307 Lens
CNRS UMR 8188, F-62307 Lens

youssefh@microsoft.com {jabbour,piette,sais}@cril.fr

Résumé

La résolution parallèle du problème SAT est maintenant un important champ de recherche. Le vaste déploiement de plateformes multi-cœurs combinés à la disponibilité d'instances SAT difficiles sont probablement à l'origine de ce succès. Toutefois, les solveurs SAT parallèles proposés jusqu'à présent souffrent d'un comportement non déterministe. Ceci est la conséquence de leur architecture qui n'effectue que peu (ou pas) de synchronisation, dans le but d'optimiser les performances. Ce caractère non déterministe est un revers pour les utilisateurs, et nombre d'applications nécessitent la reproductivité des résultats, à la fois en terme de temps d'exécution et de solutions retournées. Dans cet article, nous proposons le premier solveur SAT parallèle et déterministe (modulo le nombre de cœurs disponibles), basé sur un portfolio de solveurs SAT synchronisés. Nos résultats expérimentaux montrent clairement que notre approche préserve les performances d'un solveur non déterministe tout en garantissant une reproductibilité complète des résultats.

1 Introduction

La résolution parallèle du problème SAT a reçu énormément d'attention ces 3 dernières années. Ce phénomène est dû à plusieurs facteurs, comme la large disponibilité d'architectures multi-cœurs, ou la relative stagnation des performances des solveurs SAT séquentiels. Malheureusement, la supériorité des solveurs parallèles vient au prix de résultats non reproductibles, à la fois en terme de temps d'exécution et de solutions retournées. Ce comportement est la conséquence de leur architecture qui n'effectue aucune synchronisation dans l'objectif de maximiser les performances. Aujourd'hui, SAT est la pierre angulaire de nombreuses applications industrielles, telle que la vérification formelle, la démonstration automatique ou la bioinformatique. Ces domaines nécessitent la reproductibilité des résultats, et ne peuvent donc tirer parti de l'efficacité des solveurs parallèles. A titre d'exemple, un outil de vérification logiciel basé sur un solveur SAT parallèle peut retourner différents bugs s'il est exécuté plusieurs fois sur un même morceau de code source ; une situation inacceptable pour un ingénieur spécialisé en vérification.

Considérons un autre exemple : un utilisateur d'une application de conception d'emploi du temps basé un solveur SAT parallèle peut être étonné de voir générer des emplois du temps différents, avec pourtant le même ensemble de contraintes, via des appels successifs du solveur. Avec de telles applications, la reproductivité est préférable, sinon nécessaire. En outre, l'intégration de composants non déterministes tels qu'un solveur parallèle dans un logiciel augmente inévitablement le coût de la phase de débuggage. Ceci limite clairement le déploiement des technologies de parallélisation pour SAT.

Plus largement, en science comme en ingénierie, la re-

productibilité des résultats est fondamentale, et dans le but de servir à un nombre toujours plus grand d'applications, les solveurs parallèles doivent donc être déterministes. En effet, évaluer ou comparer des solveurs SAT non déterministes est clairement problématique. Ce problème a conduit les organisateurs des compétitions SAT et SAT-Race à mentionner clairement leur préférence pour les solveurs déterministes et d'introduire des règles spécifiques pour l'évaluation des solveurs SAT parallèles. On peut par exemple trouver ceci dans le règlement de la SAT-Race 2008 : « dans le but d'obtenir des résultats reproductibles, les solveurs SAT doivent s'abstenir autant que possible d'utiliser des constructions non déteministes. Pour les implantations parallèles, il est pourtant difficile d'obtenir des temps d'exécutions reproductibles. Ainsi, cette condition ne s'applique pas à la catégorie parallèle. Comme les temps d'exécution des solveurs paralèlles peuvent très sensiblement varier, chaque solveur est exécuté trois fois sur chaque instance. Une instance est considérée résolue si au moins l'une des trois exécutions du solveur a permis de conclure quant à sa satisfaisabilité ». Pour les catégories parallèles, des règles d'évaluation différentes ont été utilisées lors de la compétition SAT 2009, ainsi que lors de la SAT Race 2010, illustrant les problèmes inhérents au non déterminisme dans l'évaluation des solveurs SAT parallèles.

Dans ce papier, nous proposons un solveur SAT à la fois parallèle et déterministe. Ces résultats sont intégralement reproductibles, à la fois en terme de temps d'exécution et de solutions retournées, et son efficicacité est équivalente à un solveur parallèle non déterministe. Celui-ci est basé sur un contrôle des intéractions entre les différents fils d'exécution à travers l'utilisation de barrières de synchronisation. Pour maximiser son efficacité, l'échange d'informations (clauses conflits) et la vérification de terminaison sont effectués sur une base régulière. Comme nous le montrons plus loin, la fréquence de ces échange influence largement les performances de notre solveur parallèle. Ce papier explore le compromis nécessaire entre de fréquentes synchronisations, qui permettent une intégration rapide des clauses apprises par les autres fils d'exécution au prix d'une attente importante, et la synchronisation plus sporadique, qui évite le coût élevé de l'attente entre fils d'exécutions mais retarde l'importation de clauses conflits.

Ce papier est organisé comme suit : dans la section suivante, nous détaillons les caractéristiques principales des solveurs SAT séquentiels et parallèles, et fournissons la motivation de ce travail dans la section 3. Notre contribution principale est détaillée dans la section 4, et est expérimentalement évaluée dans la section 5. Enfin, nous concluons par quelques pistes intéressantes liées à ce travail.

2 Préliminaires

2.1 Solveurs SAT modernes

Le problème SAT consiste à trouver une affectation à chaque variable d'une formule propositionnelle exprimée sous forme normale conjonctive (CNF). La plupart des solveurs modernes sont basés sur une version avancée de la procédure de Davis, Putnam, Logemann and Loveland, appelée communément DPLL [9]. Un solveur SAT à la DPLL effectue une recherche arborescente, selectionnant à chaque nœud de l'arbre un littéral *de décision* qu'il affecte à une valeur booléenne. Cette affectation est suivie d'une étape d'inférence qui déduit et propage certains littéraux, dits *unitaires*. Ces opérations sont enregistrées dans un *graphe d'implication*, une structure de données centrale qui stocke l'affectation partielle courante, ainsi que ses implications unitaires. Ce *processus de branchement* est répété jusqu'à ce que soit un modèle soit trouvé, soit un conflit soit atteint. Dans le premier cas, la formule est prouvée satisfaisable, et le modèle généré est retourné, alors que dans l'autre cas, une *clause conflit* (également appelée *clause assertive*) est générée par résolution, suivant un parcours du graphe d'implication [16, 20]. Ce *processus d'apprentissage* s'arrête dès qu'une clause ne contenant qu'un seul littéral non affecté (par rapport au niveau de décision courant) est obtenue. Une telle clause conflit exprime le fait que ce littéral est impliqué à un niveau supérieur de l'arbre. Le solveur effectue donc un saut en arrière (backtrack) à ce niveau d'implication et affecte ce littéral de manière à le satisfaire. Si une clause conflit vide est générée, alors la formule est prouvée insatisfaisable.

Au-delà de ce schéma de base, les solveurs s'appuient sur des techniques additionnelles, telles que les heuristiques basées sur l'activité, ou les potiliques de redémarrage. Détaillons un peu ces deux caractéristiques importantes. L'activité de chaque variable rencontrée lors du processus d'apprentissage est incrémentée. La variable ayant la plus importante activité est sélectionnée comme prochaine variable de décision. Ceci correspond à l'heuristique de choix de variable VSIDS (Variable State Independent Decaying Sum) [20]. En outre, après qu'un certain nombre de conflits soit atteint, la recherche est redémarrée. L'évolution de cette limite de conflits est contrôlée par une politique de redémarrage (voir par exemple [1]). Grâce à l'activité des variables enregistrées durant les précédentes exécutions partielles, l'objectif du redémarrage est de se concentrer tôt dans la recherche (i.e. en haut de l'arbre de recherche) sur des parties importantes de la formule. Ceci diffère des objectifs initiaux de la randomisation et des redémarrages introduits dans [11]. Pour un panorama plus exhaustif des techniques actuelles pour la résolution de SAT, nous conseillons la lecture de [4].

2.2 Solveurs SAT parallèles

Il existe principalement deux approches de résolution parallèle en programmation par contraintes. La première implante l'idée de « diviser pour régner », qui fractionne incrémentalement l'espace de recherche en sous-espaces, successivement alloués à différents travailleurs séquentiels [8, 17, 7, 10, 15, 5]. Ces travailleurs coopérent à travers une stratégie qui rééquilibre et réalloue dynamiquement les sous-espaces aux travailleurs à l'arrêt, ainsi que par l'échange de nogood [18, 6, 7].

L'approche à base de portfolio a été introduite en 2008 pour le problème SAT. Elle exploite la complémentarité entre plusieurs recherches DPLL séquentielles concurrentes et les fait coopérer. Puisque dans ce cas chaque recherche parallèle est effectuée sur la même formule, il n'y a pas besoin d'implanter des stratégies de réallocation dynamique du travail, car la coopération n'est effectuée qu'à travers l'échange de clauses. Avec ce type d'approches, un paramétrage fin des stratégies de recherche est essentiel, plus encore avec un faible nombre d'unités de calcul. De manière générale, l'objectif est de couvrir l'espace des *bonnes* stratégies de recherche de la meilleure façon possible.

Durant ces deux dernières années, les solveurs SAT parallèles basés sur des portfolios sont devenus proéminents ; à notre connaissance, durant cette période, aucune approche de type « diviser pour règner » n'a été développée (la dernière en date étant [7]). Nous décrivons ici les différents solveurs parallèles qualifiés lors de la SAT Race 2010 [1]. Ces implantations représentent d'état de l'art en terme de solveurs SAT parallèles.

Le vainqueur de la catégorie parallèle de la SAT Race 2010, appelé `plingeling` [3], dupplique l'instance à résoudre via un fil maître et l'alloue à des fils travailleurs. Les stratégies des travailleurs se différencient principalement par la quantité de prétraitement, la graîne aléatoire et l'heuristique de choix de variables. Le partage de clauses est réduit aux clauses unitaires qui sont échangées à travers le fil maître.

`ManySAT` [12] est historiquement le premier solveur SAT portfolio. Il duplique l'instance à résoudre, et exécute différents solveurs indépendants qui se différencient par leur politique de redémarrage, leur heuristique de branchement, leur graîne aléatoire et leur technique d'apprentissage. L'échange de clauses est également effectué suivant différentes politiques. Deux versions de ce solveur ont été soumises à la SAT Race 2010, et ont terminé deuxième et troisième.

De manière similaire à `ManySAT`, le solveur `SArTagnan` [14] exploitent différentes recherches arborescentes principalement différenciées par leur politique de redémarrage, et leur heuristique inspirée de VSIDS. Certains fils appliquent un processus de résolution dynamique [2], ou exploitent la notion de « points de référence » [13]. D'autres fils essaient de simplifier la base de clauses apprises en effectant des techniques d'élimination de variables. Ce solveur a fini quatrième.

Enfin, le solveur `Antom` [19] différencie ses fils d'exécution par l'heuristique de choix de variable, la stratégie de redémarrage, la détection de clause conflit et l'hyper résolution binaire effectuée de manière paresseuse [2], ainsi que la propagation unitaire dynamique. Le partage de clauses conflits est implémentée. Ce solveur a fini cinquième.

1. `http://baldur.iti.uka.de/sat-race-2010`

3 Motivations

Comme mentionné plus haut, tous les solveurs SAT parallèles souffrent d'un comportement non déterministe. Ceci est la conséquence de leur architecture qui n'effectue que peu (ou pas) de synchronisation, dans le but de maximiser les performances. Le partage de clauses conflits, généralement utilisé pour améliorer leurs performances, accentue plus encore ce comportement non déterministe, c'est-à-dire la non-reproductibité en terme de temps d'exécution, de solution retournée ou de preuve d'insatisfaisabilité.

Pour illustrer les variations en terme de solutions et de temps d'exécution, nous avons exécuté à de multiples reprises le même solveur parallèle (`ManySAT` version 1.1 [12]) sur toutes les instances satisfaisables utilisées pour la SAT Race 2010, et nous reportons certaines mesures. Chaque instance a été résolue 10 fois, et le temps limite pour chacune des exécutions a été fixé à 40 minutes. Les résultats obtenus sont reportés dans la Table 2.2. Pour chaque instance, nous donnons le nombre de variables ($\#vars$), le nombre d'exécutions réussies ($\#models$) et le nombre de modèles différents (*diff*) obtenus par ces exécutions. Deux modèles sont différents si leur distance de Hamming n'est pas égale à 0. Nous reportons également la distance moyenne de Hamming normalisée $n\bar{H} = \frac{\bar{H}}{\#vars}$ où $\bar{H}$ est la distance de Hamming moyenne entre chaque paire de modèles obtenus.

Comme on pouvait s'y attendre, le solveur fait preuve d'un comportement non déterministe en terme de modèles retournés. En effet, sur de nombreuses instances, il est possible d'observer que chaque exécution conduit à un modèle différent. Ces différences sont illustrées dans de nombreux cas par une importante valeur pour $n\bar{H}$. Par exemple, pour la famille $sha0\_*\_*$, le nombre de modèles différents est de 10, et $n\bar{H}$ vaut environ 30%, signifiant que les modèles diffèrent de 30% de leur valeur de vérité en moyenne. Enfin, le temps moyen (tpsMoy) et la déviation standard (σ) montrent la variabilité du solveur parallèle en terme de temps d'exécution. L'instance $12pipe\_bug8$ illustre un cas extrême de ce phénomène ; en effet, pour l'obtention d'un

Instance	#vars	#models (diff)	nH	temps moyen (σ)
12pipe_bug8	117526	10 (1)	0	2,63 (53,32)
ACG-20-10p1	381708	10 (10)	1.42	1452,24 (40,61)
AProVE09-20	33054	10 (10)	33,84	19,5 (9,03)
dated-10-13-s	181082	10 (10)	0,67	6,25 (9,30)
gss-16-s100	31248	10 (1)	0	38,77 (18,75)
gss-19-s100	31435	10 (1)	0	441,75 (35,78)
gss-20-s100	31503	10 (1)	0	681 (58,27)
itox_vc1138	150680	10 (10)	26,62	0,65 (22,99)
md5_47_4	65604	10 (10)	34,8	173,9 (31,03)
md5_48_1	66892	10 (10)	34,76	704,74 (74,65)
md5_48_3	66892	10 (10)	34,16	489,02 (68,96)
safe-30-h30-sat	135786	10 (10)	22,32	0,37 (0,79)
sha0_35_1	48689	10 (10)	33,18	45,4 (21,88)
sha0_35_2	48689	10 (10)	33,25	61,65 (29,93)
sha0_35_3	48689	10 (10)	32,76	72,21 (21,93)
sha0_35_4	48689	10 (10)	33,2	105,8 (35,22)
sha0_36_5	50073	10 (10)	34,19	488,16 (58,58)
sortnet-8-ipc5-h19-sat	361125	4 (4)	15,86	2058,39 (47,5)
total-10-19-s	331631	10 (10)	0,5	5,31 (6,75)
UCG-20-10p1	259258	10 (10)	2,12	768,17 (31,63)
vmpc_27	729	10 (2)	2,53	11,95 (32,62)
vmpc_28	784	10 (2)	3,67	34,61 (25,92)
vmpc_31	961	8 (1)	0	583,36 (88,65)

TABLE 1 – Comportement non déterministe de `ManySAT`

Algorithme 1 : DPLL Deterministe Parallèle

Données : une formule CNF $\mathcal{F}$;
Résultat : $vrai$ si $\mathcal{F}$ est satisfaisable ; $faux$ sinon

1 **début**
2 $<en\ parallele, 0 \leq i < \#coeur>$
3 reponse[i] = search($coeur_i$) ;
4 **pour** *($i = 0$; $i < \#coeur$; i++)* **faire**
5 **si** *($reponse[i]! = inconnu$)* **alors**
6 **retourner** reponse[i];
7 **fin**

même et unique modèle (*diff* =1), la déviation standard du temps d'exécution est d'environ 53 secondes alors que sa moyenne est de 2,63 secondes. Cette première expérimentation illustre clairement à quel point le comportement non déteministe des solveurs SAT parallèles influence la non-reproductibilité des résultats retournés.

4 DPLL Déterministe Parallèle : $(DP)^2LL$

Dans cette partie, nous présentons le premier solveur SAT parallèle et déteministe. Comme le partage de clauses a été prouvé comme étant un élément important dans l'efficacité des techniques de résolution parallèle, notre but est de concevoir une approche déterministe tout en conservant ce partage de clauses. Dans ce but, notre technique de déterminisation est basée sur l'introduction de deux barrières de synchronisation (notées $< barriere >$) qui représentent des points de l'algorithme auxquels les différents fils d'exécution s'attendent les uns les autres. Ces barrières sont placées pour synchroniser à la fois le partage de clauses et la détection de la terminaison de la procédure (cf section 4.1). Par la suite, nous proposons une technique permettant d'ajuster dynamiquement la période de synchronisation avant que les fils ne se rejoignent à la barrière (section 4.2).

4.1 Déterminisation statique

Décrivons maintenant notre approche de déterminisation de solveurs parallèles basés sur un portfolio. Rappelons tout d'abord qu'un solveur basé sur un portfolio exécute différentes instances d'une procédure de type DPLL sur le même problème. Les lignes 2 et 3 de l'algorithme 1 illustrent cet aspect portfolio, en lançant en parallèle plusieurs recherches sur les cœurs de calcul disponibles. Pour éviter tout non-déterminisme en terme de solution retournée ou de preuve d'insatisfaisabilité, une structure de don-

Algorithme 2 : search(coeur$_i$)

```
   Données : une formule CNF F;
   Résultat : reponse[i] = vrai si F est satisfaisable ;
              faux si F is insatisfaisable, inconnu sinon
1  début
2      nbConflicts=0;
3      tant que (vrai) faire
4          si ( !propage()) alors
5              nbConflicts++;
6              si (topLevel) alors
7                  reponse[i]= faux ;
8                  goto barriere_1;
9              learntClause=analyse();
10             exporteExtraClause(learntClause);
11             backtrack();
12             si (nbConflicts % periode == 0) alors
13                 barriere_1 : <barriere>
14                 si (∃j|reponse[j]! = inconnu) alors
15                     retourner reponse[i] ;
16                 miseAJourPeriode();
17                 importExtraClauses();
18                 <barriere>
19         sinon
20             si ( !decide()) alors
21                 reponse[i]= vrai ;
22                 goto barriere_1;
23 fin
```

née globale nommée $reponse$ est utilisée pour enregistrer l'état des différents fils d'exécutions sur la satisfaisabilité de l'instance. Les fils sont triés par rapport à leur identifiant (un entier, de 0 to #coeur-1). L'algorithme 1 retourne le résultat obtenu par le premier cœur (dans cet ordre) si plusieurs d'entre eux peuvent conclure quant à la satisfaisabilité de la formule (lignes 4-6). Ceci est une condition nécessaire mais non suffisante pour la reproductibilité des résultats. Pour parvenir à une déterminisation complète du solveur parallèle, jetons un œil plus attentif au cœur de l'algorithme DPLL alloué à chaque fil (Algorithme 2). En plus des composants habituels des solveurs de type CDCL, on peut observer l'introduction de deux barrières de synchronisation (<*barriere*>, lignes 13 et 18). Pour comprendre le rôle de ces points de synchronisation, nous avons besoin de considérer à la fois leur place au sein de l'algorithme et la région circonscrite par ces deux barrières. Tout d'abord, la première barrière étiquetée $barriere_1$ (ligne 13) est placée avant que tout fil d'exécution puisse retourner l'état de la satisfaisabilité de la formule. Cette barrière les empêche de retourner leur propre solution de manière anarchique, mais les force au contraire à s'attendre entre eux (lignes 14 et 15). Ceci explique pourquoi la procédure est renvoyée à cette barrière dès que l'insatisfaisabilité est prouvée (backtrack au niveau 0 de l'arbre de recherche, lignes 6-8), ou quant un modèle est obtenu (lignes 20-22). A la ligne 14, si la satisfaibilité de la formule est prouvée par l'un des fils ($\exists coeur_j|reponse[j]! = inconnu$), l'algorithme retourne son propre $reponse[i]$. Si aucun ne peut répondre, ils partagent alors de l'information en important les clauses conflits générées par les autres fils durant la dernière *période* (ligne 17). Après que chacun des fils aient fini l'importation (deuxième barrière, ligne 18), ils poursuivent leur exploration, à la recherche d'une solution. Cette seconde barrière est intégrée de manière à empêcher les fils de quitter la région de synchronisation avant les autres. En d'autres termes, quand l'un des fils atteint cette deuxième barrière, il doit attendre que les autres fils aient terminé l'importation des clauses étrangères avant de poursuivre. Comme différents ordres d'importation des clauses peuvent induire différents ordres au niveau de la propagation unitaire, et par conséquent différents arbres de recherche, les clauses apprises sont importées (ligne 17) suivant un ordre fixé par les fils en fonction de leur identifiant.

Pour compléter cette présentation, nous spécifions certaines fonctions usuelles de la procédure de recherche. Tout d'abord, la fonction $propage()$ (ligne 4) applique la propagation unitaire et retourne *faux* si un conflit survient, *vrai* sinon. Dans le premier cas, une clause est apprise par la fonction $analyse()$ (ligne 9), et est ajoutée à la formule puis exportée vers les autres fils (ligne 10, fonction $exporteExtraClause()$). Ces clauses apprises sont périodiquement importées dans la région synchronisée (ligne 17). Dans le cas contraire, la fonction $decision()$ choisit la prochaine variable à affecter et retourne *vrai* ; si elle ne retourne *faux*, alors toutes les variables sont d'ores et déjà affectées et un modèle a été trouvé.

Pour maximiser la dynamique d'échange d'information, chaque fil peut se synchroniser avec les autres avec chaque conflit, en important toute clause apprise juste après sa génération. Malheureusement, comme nous le verrons dans les parties suivantes, cette solution est très inefficace en pratique, puisque beaucoup de temps est perdu par les fils d'exécution à s'attendre entre eux. Pour atténuer ce problème, nous proposons de ne synchroniser les fils qu'après un certain nombre de conflits, noté $periode$ (ligne 10). Cette approche, notée $(DP)^2LL\_static(periode)$, ne met pas à jour la période pendant la recherche (pas d'appel à la fonction $miseAJourPeriode()$, ligne 16). Toutefois, bien qu'une valeur optimale puisse être obtenue pour le paramètre $periode$, le problème de l'attente aux barrières ne peut être complètement éliminé. En effet, alors que les différents fils présentent des comportements de recherche différents (stratégie de recherche, taille de la base de clauses apprises, etc.) utiliser la même valeur de $periode$ pour les différents fils conduit inévitablement à gâcher du temps à la première barrière.

Dans la partie suivante, nous proposons une technique de synchronisation dynamique, qui exploite la taille de la formule courante pour déduire une valeur spécifique du paramètre $periode$ pour chaque fil (appel à la fonction $miseAJourPeriode()$, ligne 16).

4.2 Synchronisation dynamique

Comme mentionné plus haut, les procédures de type DPLL exécutées par les différents fils développent différents arbres de recherche. En conséquence, les temps d'attente ne peuvent être évités même si une valeur statique est optimalement définie. Dans cette section, nous proposons une synchronisation dynamique de la valeur du paramètre $periode$. Notre but est de réduire autant que possible le temps gâché en attente à la barrière. Or, le temps nécessaire à chaque fil pour effectuer le même nombre de conflit est difficile à estimer à l'avance. Nous proposons néanmoins une mesure d'approximation, qui estime le coût calculatoire de la propagation unitaire pour en déduire une vitesse de progression de chaque fil. Plus précisément, puisque les différents fils sont exécutés sur la même formule, nous utilisons la taille de la base de clauses apprises courante comme estimation de cette opération basique. Ainsi, notre synchronisation dynamique est calculée en fonction de ces informations.

Décrivons formellement notre approche. Dans notre stratégie de synchronisation, pour chaque unité de calcul u_i, nous considérons une séquence de temps de synchronisation comme une séquence d'étapes t_i^k avec $t_i^0 = 0$ et $t_i^k = t_i^{k-1} + periode_i^k$ où $periode_i^k$ représente la fenêtre de temps définie en terme de nombre de conflits entre t_i^{k-1} et t_i^k. Cette séquence est évidemment différente pour les unités de calculs u_i ($0 \leq i < \#coeur$). Nous définissons Δ_i^k comme l'ensemble de clauses apprises par u_i à l'étape t_i^k. Dans la suite, quand il n'y a aucune ambigüité, nous noterons k plutôt que t_i^k.

Soit $m = max_{\forall u_i}(|\Delta_i^k|)$, où $0 \leq i < \#coeur$ est la taille de la plus grande base de clauses apprises et $S_i^k = \frac{|\Delta_i^k|}{m}$ le ratio entre la taille de clauses apprises de u_i et m. Ce ratio S_i^k représente la vitesse estimée de u_i. Quand ce ratio tend vers 1, la progression de u_i est proche du fil qui est heuristiquement le plus lent, tandis que s'il tend vers 0, le fil u_i progresse beaucoup plus vite que le plus lent. Pour $k = 0$ et pour chaque u_i, nous définissons $periode_i^0$ à α, où α est un entier naturel. Ensuite, à l'étape $k > 0$, et pour chaque u_i, la prochaine valeur est calculée comme suit : $periode_i^{k+1} = \alpha + (1 - S_i^k) \times \alpha$, où $0 \leq i < \#coeur$. Intuitivement, le fil ayant la plus grande valeur pour S_i^k (tendant vers 1) doit avoir la période la plus courte. Au contraire, le fil avec le plus petit S_i^k (tendant vers 0) doit avoir la période la plus longue.

5 Évaluation expérimentale

Toutes nos expérimentations ont été conduites sur des processeurs Intel Xeon 3GHz sous Linux CentOS 4.1. (noyau 2.6.9) avec une limite de mémoire vive de 2 Go. Notre algorithme DPLL déteministe a été implanté sur le solveur parallèle basé un portfolio `ManySAT` (version 1.1). Le temps limite a été fixé à 900 secondes pour chaque instance, et si aucune réponse n'a été apportée dans cette limite de temps, l'instance est considérée non résolue. Nous avons utilisé les 100 instances proposées lors de la récente `SAT Race 2010`, et nous reportons pour chaque expérimentation le nombre d'instances résolues (x-axis) avec le temps nécessaire à leur résolution (y-axis). Chaque solveur parallèle est exécuté en utilisant quatre fils d'exécution. Notons que dans les expérimentations suivantes, plutôt que le temps CPU utilisé habituellement, nous avons préféré considéré le temps réel. En effet, dans la plupart des architectures, le temps CPU n'est pas incrémenté quand les fils d'exécution sont endormis (temps d'attente à la barrière). Ainsi, prendre en considération le temps CPU donnerait à nos implantations déterministes un avantage à la fois substantiel et illégitime.

5.1 Période statique

Dans cette première expérimentation, nous avons évalué les performances de notre algorithme **DPLL D**éterministe et **P**arallèle ($(DP)^2$LL) avec différentes périodes de synchronisation. La Figure 1 présente les résultats obtenus. Tout d'abord, une version séquentielle du solveur a été exécutée. Sans surprise, cette version obtient les pires résultats globaux en ne résolvant que 68 instances en plus de 11 000 secondes. Ce résultat permet d'apprécier le gain d'obtenu par les mécanismes parallèles. Nous reportons également les résultats obtenus par l'exécution de la version non déterministe du solveur `ManySAT 1.1`. Soulignons que, comme présenté dans la section 3, lancer plusieurs fois ce solveur peut conduire à des résultats différents. Ce solveur non déterministe a permis la résolution de 75 instances en 8 850 secondes. Ensuite, nous avons exécuté une version déteministe de ce même solveur, dans laquelle chaque fil se synchronise après la génération de chaque clause conflit ((DP)2LL_static(1)). Il est possible d'observer le poids conséquent, d'un point de vue calculatoire, de cette barrière. En effet, cette version est clairement moins efficace que la version non déterministe, en ne résolvant que 72 instances sur les 100 proposée, et cela en plus de 10 000 secondes.

Ce résultat négatif est principalement dû au temps perdu par les fils d'exécution à s'attendre les uns les autres de manière très fréquente. Pour pallier ce problème, nous avons également essayé de ne synchroniser les fils qu'après un certain nombre de conflits (100, 10 000). La Figure 1 montre que ces versions surpassent en efficacité la version

periode	**temps de résolution**	**temps d'attente**	**attente/résolution (en %)**
static(1)	10 276	4 208	40,9
static(100)	9 715	2 559	26,3
static(10000)	9 124	1 605	17,6

TABLE 2 – Temps d'attente aux barrières de synchronisation par rapport à la période

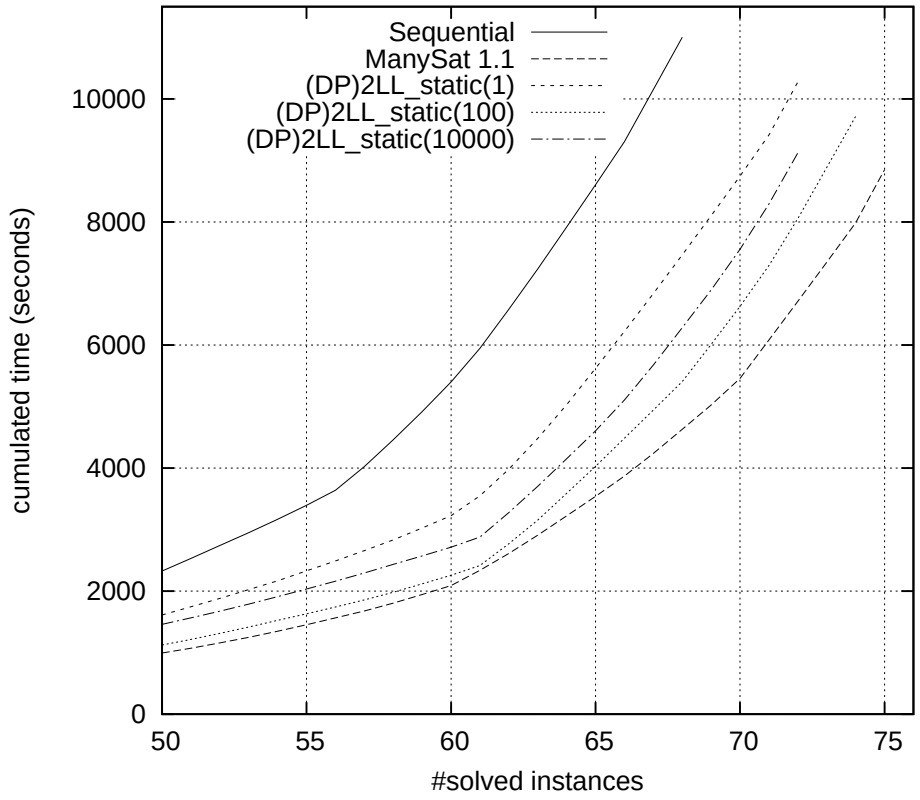


FIGURE 1 – Performances avec synchronisation statique

« periode=1 », mais elles restent loin des résultats obtenus avec la version non déterministe.

Ces deux barrières permettant à notre algorithme d'être déterministe sont clairement un inconvénient à son efficacité. Ainsi, nous avons évalué la quantité de temps passé par les fils d'exécution à s'attendre les uns les autres. Dans la table 2, nour fournissons le rapport de temps passé à attendre dans un but de synchronisation sur le temps d'exécution total pour les différentes versions de `(DP)2LL`. Comme les résultats le montrent, ce temps peut être très significatif quant l'attente est faite après chaque conflit. En effet, dans ce cas, plus de 40% du temps total est passé par les fils d'exécution à s'attendre entre eux. Comme nous le prévoyions, ce temps est réduit quand la synchronisation ne survient que plus épisodiquement, mais n'est jamais négligeable. Notons que bien que la version « periode=10000 » perd moins de temps à attendre que la version« periode=100 », elle obtient les pires résultats globaux en ne résolvant que 72 instances. La version « periode=100 » obtient les meilleurs résultats (si on exclut la version non déterministe). Après des expérimentations intensives, ce paramètre semble être un bon compromis entre le temps d'attente à la barrière et la dynamique d'échange d'informations. En effet, nous pensons que pour la version « periode=10000 », l'information est échangée trop tard, ce qui explique son comportement peu satisfaisant.

De ce fait, nous avons tenté de réduire le temps que les fils d'exécution passent à attendre à la barrière en adoptant une stratégie dynamique (présentée dans section 4.2). Les résultats empiriques concernant cette version sont présentées dans la partie suivante.

5.2 Période dynamique

Dans une seconde expérimentation, nous avons evalué notre stratégie dynamique. Nous comparons les résultats de cette version contre ceux obtenus par `ManySAT 1.1`, et

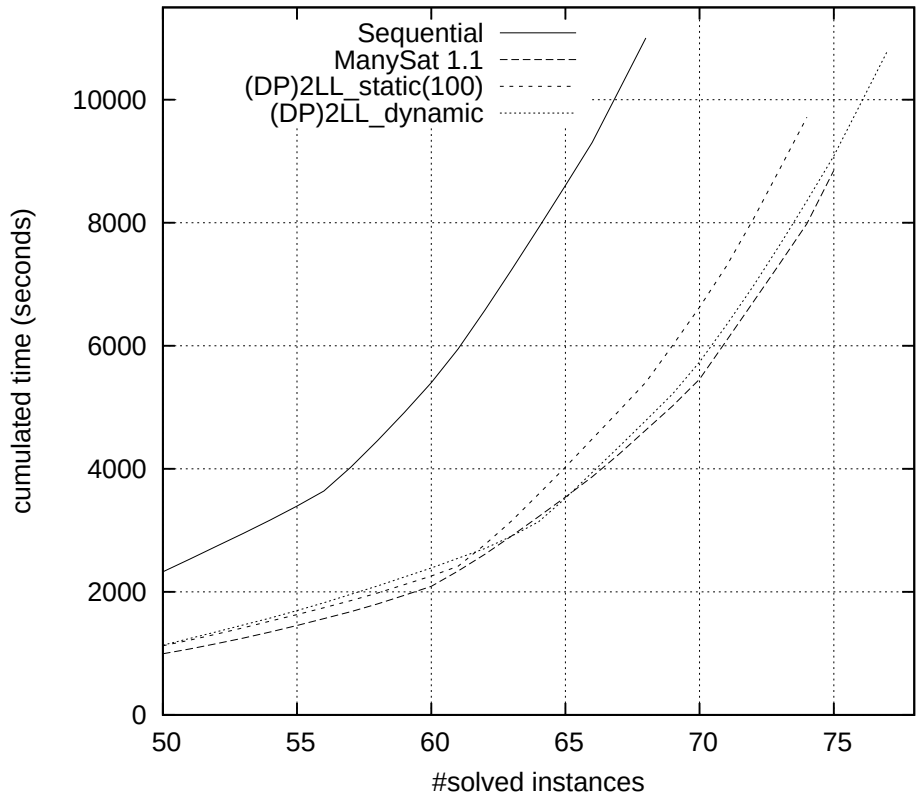


FIGURE 2 – Performances avec synchronisation dynamique

fournissons également les résultats de la meilleure version statique (100), et de la version séquentielle. Les résultats obtenus sont reportées dans la Figure 2. Cette version dynamique est exécutée avec le paramètre $\alpha = 100$.

Cette expérimentation confirme l'intuition suivant laquelle chaque fil d'exécution doit avoir une période de synchronisation différente, suivant la taille de leur base de clauses apprises, permettant d'estimer heuristiquement sa vitesse de propagation. En effet, on peut observer dans la Figure 2 que la « courbe de résolution » de cette version dynamique est très proche de celle `ManySat 1.1`. Ceci montre que les 2 solveurs sont capables de résoudre à peu près le même nombre d'instance dans des temps similaires. De plus, cette version adaptative est capable de résoudre 2 instances de plus que la version non déterministe, ce qui en fait la version la plus efficace testée pendant nos expérimentations.

6 Discussion

Dans cet article, nous apportons une première réponse au problème important du non-déteminisme que pose les solveurs SAT parallèles, en proposant $(DP)^2LL$ une première implantation d'une procédure déterministe parallèle. A cette fin, une idée simple mais efficace est présentée : elle consiste principalement en l'introduction de barrières de synchronisation à l'algorithme originel. Nous proposons plusieurs stratégies de synchronisation, et montrons que cette version déterministe se montre empiriquement très efficace. En effet, ses résultats sont similaires à une version non déterministe.

Cette première implantation ouvre de nombreuses perspectives de recherche. Tout d'abord, comme dans la plupart des procédures parallèles basées sur l'échange de clauses, le nombre d'unités de calcul est un goulot d'étranglement pour l'efficacité de la recherche. Exécuter des solveurs sur un grand nombre d'unités de calcul (par exemple 32) ne garantit pas que de meilleurs résultats seront obtenus. En fait, comme les clauses sont échangés par un plus grand nombre de travailleurs, les techniques actuelles sont souvent moins efficaces quand elles sont utilisées sur un grand nombre d'unités de calculs. Dans ce papier, nous avons proposé une approche garantissant le déterminisme aux solveurs SAT parallèles. Une autre caractéristique clé à ajouter à ces solveurs serait l'opportunisme par rapport aux ressources utilisées (nombre d'unités de calcul, mémoire, etc.). Nous prévoyons d'étudier cela dans un travail futur. Ensuite, les barrières ajoutées à la boucle principale de l'algorithme CDCL parallèle peuvent être vues comme un revers à l'efficacité pratique de la procédure. En effet, chaque fil d'exécution ayant terminé son calcul partiel doit attendre que les autres fils aient également terminé. Néanmoins, notre

politique de synchronisation se montre très efficace tout en conservant le cœur de l'architecture parallèle : l'échange de clauses. De nouvelles techniques permettant aux fils d'exécution d'interagir peuvent être imaginées à cet endroit particulier de la recherche. Nous pensons que ceci est une perspective de recherche très intéressante.

Références

[1] A. Biere. Adaptive restart strategies for conflict driven SAT solvers. In *SAT*, pages 28–33, 2008.

[2] A. Biere. Lazy hyper binary resolution. Technical report, Dagstuhl Seminar 09461, 2009.

[3] A. Biere. Lingeling, plingeling, picosat and precosat at SAT race 2010. Technical Report 10/1, FMV Reports Series, 2010.

[4] A. Biere, M. J. H. Heule, H. van Maaren, and T. Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2009.

[5] Max Böhm and Ewald Speckenmeyer. A fast parallel sat-solver - efficient workload balancing. *Annals of Mathematics and Artificial Intelligence*, 17(3-4) :381–400, 1996.

[6] W. Chrabakh and R. Wolski. GrADSAT : A parallel SAT solver for the grid. Technical report, UCSB Computer Science, 2003.

[7] G. Chu and P. J. Stuckey. Pminisat : a parallelization of minisat 2.0. Technical report, SAT Race, 2008.

[8] Geoffrey Chu, Christian Schulte, and Peter J. Stuckey. Confidence-based work stealing in parallel constraint programming. In *15th International Conference on Principles and Practice of Constraint Programming - CP 2009*, pages 226–241, 2009.

[9] M. Davis, G. Logemann, and D. W. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7) :394–397, 1962.

[10] Luis Gil, Paulo Flores, and Luis Miguel Silveira. PMSat : a parallel version of minisat. *Journal on Satisfiability, Boolean Modeling and Computation*, 6 :71–98, 2008.

[11] C. Gomes, B. Selman, and H. Kautz. Boosting combinatorial search through randomization. In *AAAI*, pages 431–437, 1998.

[12] Y. Hamadi, S. Jabbour, and L. Sais. ManySAT : a parallel SAT solver. *Journal on Satisfiability, Boolean Modeling and Computation*, 6 :245–262, 2009.

[13] S. Kottler. SAT solving with reference points. In *SAT*, pages 143–157, 2010.

[14] S. Kottler. SArTagnan : solver description. Technical report, SAT Race 2010, July 2010.

[15] M. Lewis, T. Schubert, and B. Becker. Multithreaded sat solving. In *12th Asia and South Pacific Design Automation Conference*, 2007.

[16] J. Marques-Silva and K. Sakallah. GRASP - A New Search Algorithm for Satisfiability. In *ICCAD*, pages 220–227, 1996.

[17] Laurent Michel, Andrew See, and Pascal Van Hentenryck. Transparent parallelization of constraint programming. *INFORMS Journal on Computing*, 21(3) :363–382, 2009.

[18] Carl Christian Rolf and Krzysztof Kuchcinski. Load-balancing methods for parallel and distributed constraint solving. In *Proceedings of the 2008 IEEE International Conference on Cluster Computing*, pages 304–309, 2008.

[19] T. Schubert, M. Lewis, and B. Becker. Antom : solver description. Technical report, SAT Race, 2010.

[20] L. Zhang, C. Madigan, M. Moskewicz, and S. Malik. Efficient conflict driven learning in boolean satisfiability solver. In *ICCAD*, pages 279–285, 2001.

Backtracking adaptatif

Olivier Lhomme

IBM France

om.lhomme@gmail.com ou olivier.lhomme@fr.ibm.com

Abstract

L'algorithme de backtracking diffère de la recherche en profondeur d'abord par l'utilisation, à chaque noeud, d'une procédure de filtrage capable, dans certains cas, de détecter l'absence de solution parmi les descendants du noeud. L'exploration du sous-arbre correspondant peut ainsi etre évitée. Pour un domaine d'application particulier, il existe en général différentes procédures de filtrage. Nous proposons dans ce papier de doter l'algorithme de backtracking de la capacité de choisir, à chaque noeud, entre plusieurs algorithmes de filtrage, et en démontrons l'intérêt.

1 Introduction

L'algorithme de backtracking [1] est une composante essentielle de nombreux solveurs de problèmes combinatoires. Son principe est d'étendre la solution partielle courante, et d'explorer les différentes extensions possibles par une recherche en profondeur d'abord. Un prédicat, dit de *pruning*, capable dans certains cas de détecter qu'aucune solution complète ne peut étendre la solution partielle courante, permet à l'algorithme de backtracking d'éviter l'exploration de sous-arbres inutiles.

La capacité d'un algorithme de pruning à détecter de tels culs-de-sac est évidemment très dépendante de son temps d'exécution, ce qui signifie que le choix d'un algorithme de pruning est toujours affaire de compromis entre la taille de l'arbre de recherche exploré par l'algorithme de backtracking et le temps passé à chaque noeud de l'arbre de recherche. Par ailleurs, même si un algorithme de pruning particulier offre le meilleur compromis pour un arbre de recherche, un autre algorithme de pruning peut offrir un bien meilleur compromis localement à un sous-arbre. Cette remarque suggère de sélectionner un algorithme de pruning potentiellement différent à chaque noeud de l'arbre de recherche. Nous avons developpé cette idée, et montré qu'un choix dynamique judicieux peut améliorer les performances du backtracking de plusieurs ordres de grandeurs.

Etant donné un ensemble de prédicats de pruning, nous pouvons considérer tous les algorithmes de backtracking qui appliquent à chaque noeud l'un quelconque des ces prédicats de pruning. Nous pouvons définir le temps de calcul minimum parmi tous ces algorithmes de backtracking pour résoudre un problème donné. Ce temps de calcul théoriquement minimum peut servir de référence pour comparer la performance des algorithmes. Par exemple, étant donnés deux prédicats de pruning, le second bénéficiant d'une plus grand capacité à détecter les culs-de-sac, mais étant k fois plus lent que le premier, nous voulons savoir quel gain un algorithme de backtracking optimal peut apporter comparé aux deux algorithmes classiques avec un prédicat de pruning fixé. Nous montrons que pour certains problèmes, ce gain peut être, *à la fois*, d'un facteur exponentiel par rapport au backtracking avec le pruning faible et d'un facteur k par rapport au backtracking avec le pruning fort.

Malheureusement, un algorithme de backtracking optimal semble être irréalisable en pratique : il nécessiterait de connaître à l'avance le prédicat de pruning à appliquer à chaque noeud. Dans ce papier nous montrons que l'on peut concevoir des algorithmes de backtracking dont le coût en temps de calcul est proche de celui, purement théorique, de l'algorithme de backtracking optimal. Nous introduisons le principe de *backward-activated pruning* : au lieu d'élaguer un arbre de la racine vers les feuilles, l'idée est d'effectuer l'élagage lors d'un backtrack, c'est-à-dire des feuilles vers la racine. Nous montrons que le coût de calcul du backtracking avec le backward-activated pruning est au plus à un facteur $3 + \log_2(d-1)$ de l'exploration optimale (où d est la profondeur de l'arbre de recherche). De plus nous introduisons une propriété d'incrémentalité *bottom-up* pour les algorithmes de pruning, qui rend le backtracking avec backward-activated pruning plus efficace, avec un coût au plus à un facteur 3 du coût optimal. Les expérimentations confirment que le principe de backward-activated pruning améliore fortement l'algorithme de backtracking.

Ce papier est organisé de la manière suivante : la section 2 introduit le backtracking optimal ; la section 3 présente le backtracking avec backward-activated pruning ; la section 4 présente une analyse théorique, dont les preuves et détails sont donnés en annexe ; la section 5 résume quelques expérimentations et la section 6 aborde les travaux connexes et les perspectives.

2 Backtracking et Exploration Optimale

2.1 Préliminaires

L'algorithme de backtracking (Figure 1) commence par appliquer le prédicat *prune(c)* sur la solution partielle courante c. Si le prédicat *prune(c)* est capable de prouver que c ne peut être étendue en une solution complète, il renvoie *true*, et, donc, il est inutile d'explorer les extensions de c. Autrement, Backtracking est appelé récursivement sur les extensions possibles de c, générées par la fonction *children(c)*.

```
procedure Backtracking(c)
  if prune(c) then return
  for s ∈ children(c) do
    Backtracking(s)
```

FIG. 1 – Backtracking

Commençons par quelques définitions :

- Nous utiliserons parfois le terme *noeud* pour une solution partielle (ou complète), par référence au noeud dans l'arbre de recherche généré par l'algorithme de backtracking.
- Nous associerons un *opérateur de pruning* P au prédicat *prune* : soit N un noeud, $P(N)$ est défini comme suit :

$$P(N) = \begin{cases} \emptyset \text{ si } \textit{prune}(N) \text{ est vrai} \\ N \text{ si } \textit{prune}(N) \text{ est faux} \end{cases}$$

- Un opérateur de pruning a un *coût*, C, qui typiquement représente le temps d'exécution de l'algorithme de pruning. Nous supposerons que ce coût est constant.
- Une propriété essentielle d'un opérateur de pruning P est la *soundness* : il ne peut pas couper une branche qui conduit à une solution.
- Nous devrons comparer les capacités de filtrage des opérateurs de pruning : étant donnés deux opérateurs de pruning P_1 et P_2, nous dirons que P_2 est *plus fort que* P_1 si, pour chaque noeud N nous avons $P_1(N) = \emptyset \Rightarrow P_2(N) = \emptyset$.
- Nous étendons la fonction *children* utilisée dans l'algorithme de backtracking au symbole $\emptyset$: $children(\emptyset) = \emptyset$; nous supposerons que les solutions complètes n'ont pas de noeud fils.
- Nous supposerons qu'un opérateur de pruning P est toujours *monotone* : pour tout noeud fils N' d'un noeud N tel que $P(N) = \emptyset$, nous avons $P(N') = \emptyset$.

2.2 Backtracking Optimal

Dans cette section, nous étudions l'utilisation de plusieurs prédicats de pruning ayant différentes puissances de filtrage et différents temps d'exécution.

Considérons deux opérateurs de pruning P_1 et P_2, et deux versions de l'algorithme de backtracking : BT_1 qui utilise l'opérateur de pruning P_1, et BT_2 qui utilise P_2. Supposons que P_2 est plus fort que P_1, mais aussi que le coût de P_1 est inférieur à celui de P_2 (autrement, P_2 domine toujours P_1, et il n'y aucune raison de considérer P_1). Pour simplifier, nous supposerons que $C_2 = kC_1$ où k est un nombre entier supérieur à 1. Le coût total de l'algorithme BT_i pour explorer l'arbre de racine N peut être défini récursivement : l'opérateur P_i est appliqué sur le noeud N, ce qui coûte C_i ; puis, la récurrence sur les descendants s'applique. Nous obtenons :

$$Cost_{BT_i}(N) = C_i + \sum_{s \in children(P_i(N))} Cost_{BT_i}(s)$$

Notons les deux cas terminaux de la récurrence : (1) lorsque le noeud N est filtré, nous avons $P_i(N) = \emptyset$, et, donc, $children(P_i(N)) = \emptyset$; (2) lorsque N is une solution, N ne peut pas être filtré et n'a pas de descendant, donc nous avons $children(P_i(N)) = children(N) = \emptyset$.

Maintenant, imaginons que l'algorithme de backtracking, au lieu de toujours appliquer le même opérateur de pruning, applique parfois P_1 et parfois P_2, et considérons tous les algorithmes de cette forme possibles. Pour un problème donné, nous voudrions être aussi proche que possible du coût d'exploration minimal parmi tous ces algorithmes de backtracking. Nous appelerons *backtracking optimal*, et noterons BT_{opt}, un algorithme de backtracking idéal, capable, à chaque noeud, de sélectionner P_1 ou P_2 de telle façon que le coût total d'exploration soit minimal. Ce coût total de l'algorithme BT_{opt} pour explorer l'arbre de racine N peut être défini récursivement :

$$Cost_{BT_{opt}}(N) =$$

$$min \begin{cases} C_1 + \sum_{s \in children(P_1(N))} Cost_{BT_{opt}}(s) \\ C_2 + \sum_{s \in children(P_2(N))} Cost_{BT_{opt}}(s) \end{cases}$$

Une conséquence directe de la définition de BT_{opt} est la suivante :

Proposition 2.1 *Pour tout arbre de racine N, pour tout algorithme de backtracking A qui applique opérateur P_1 à certains noeuds et l'opérateur P_2 aux autres noeuds, $Cost_{BT_{opt}}(N) \leq Cost_A(N)$.*

```
procedure BackwardPruning(c)
1     credit := credit + C1
2     if credit < C2
3        if prune1(c) then return disablePrune2
4     else
5        credit := credit - C2
6        if prune1(c) then return enablePrune2
7        if prune2(c) then return enablePrune2
8     for s ∈ children(c) do
9        status := BackwardPruning(s)
10       if status = enablePrune2
11          if prune2(c) then return enablePrune2
12    return disablePrune2
```

FIG. 2 – Backward pruning

De plus, pour BT_1 et BT_2, nous avons la proposition suivante :

Proposition 2.2
(i) BT_{opt} *peut gagner un facteur exponentiel comparé à* BT_1.
(ii) BT_{opt} *peut gagner un facteur* C_2/C_1 *comparé à* BT_2.
(iii) De plus, BT_{opt} *peut gagner à la fois (i) et (ii) sur la* même *instance de problème.*

La preuve de cette proposition est donnée en annexe.

La proposition 2.2 fait apparaître tout l'intérêt d'un backtracking optimal pour l'exploration d'un arbre. Malheureusement, il nécessite de connaître à l'avance le prédicat de pruning à appliquer à chaque noeud, ce qui ne semble pas être possible dans le cas général. Dans la section suivante, nous présentons des algorithmes s'approchant d'une exploration optimale.

3 Backward-Activated Pruning

Dans la section 3.1, nous introduisons un algorithme de backtracking appelé *backward pruning*, puis, dans la section 3.2, nous introduisons une version dichotomique du backward pruning qui améliore sa complexité.

3.1 Backward Pruning

La figure 2 présente un algorithme, *backward pruning*, qui utilise deux algorithmes de pruning différents, $prune_1$ faible puissance de filtrage et rapide, et $prune_2$ puissance de filtrage plus forte et plus coûteux. Le filtrage faible est classiquement appliqué à chaque ouverture d'un nouveau noeud, alors que le filtrage fort est appliqué au backtrack (*backward-activated*), lorsqu'il a été détecté comme étant utile dans un sous-arbre.

Dans la fonction BackwardPruning, la variable globale *credit*, initialisée à 0, contrôle l'exécution périodique de l'algorithme de filtrage fort : à chaque appel du prédicat $prune_1$, *credit* is incrémenté de C_1, et lorsque *credit* atteint C_2, le coût de l'appel du prédicat $prune_2$, *credit* est remis à 0 (rappelons que nous supposons que C_2/C_1 est entier), et $prune_2$ est exécuté. Remarquons que $prune_2$ est appelé soit directement à ce niveau, soit, dans le cas où *prune$_1$(c)* est vrai, après backtracking.

Lorsque le crédit n'est pas suffisant pour appliquer le filtrage fort (lignes 2–3), seul le filtrage faible est appliqué. Dans ce cas, si le noeud courant est détecté comme étant un cul-de-sac, un backtrack a lieu : la fonction retourne la valeur *disablePrune2*. Lorsque le crédit est suffisant pour appliquer le filtrage fort (lignes 4–7), le filtrage faible est appliqué en premier, suivi du filtrage fort. Si l'un de ces filtrages détecte que le noeud courant est un cul-de-sac, un backtrack a lieu, la fonction retournant la valeur *enablePrune2*. Dans ce cas, le filtrage fort est aussi appliqué sur le père de noeud lors du backtrack (lignes 10–11). Les lignes 8 à 11 développent le noeud et explorent ses descendants.

Cette implémentation relativement simple de l'idée d'appliquer le filtrage fort des feuilles vers la racine sera utile pour l'analyse théorique de la section 4. Une implémentation plus réaliste éviterait l'application du filtrage fort lorsqu'elle est inutile[1].

3.2 Backward Pruning Dichotomique

Pour améliorer la complexité de l'algorithme backward pruning, nous en présentons une version dichotomique (figure 3). L'algorithme backward pruning applique le filtrage fort en remontant d'une feuille vers la racine jusqu'à trouver le plus récent ancêtre qui ne peut être élagué. Pour trouver ce même ancêtre, la version dichotomique du backward pruning utilise la méthode *findLevel*. Remarquons que cette fonction *Divide&Conquer-Backward Pruning* stocke tous les ancêtres du noeud courant dans un tableau global *node* indexé par leur profondeur dans l'arbre de recherche (voir ligne 1). Ce tableau est utilisé par la méthode *findLevel*. Une variable globale *dknp* (pour "deepest known as non prunable") maintient le niveau i le plus profond, dans la pile des appels récursifs, qui est connu comme non élaguable (*prune$_2$(node[i])* est faux). La méthode dichotomique commence par ce niveau le plus profond au lieu de 0, ce qui nous donnera une meilleure borne pour la complexité. Lors du premier appel à *Divide&Conquer-BackwardPruning*, le paramètre *level* doit être égal à 0, et la variable *dknp* doit être initialisée à -1.

4 Analyse théorique

La section 4.1 présente les résultats de l'analyse théorique des algorithmes de backward pruning : nous mon-

[1] Par exemple, lorsque le filtrage fort a déjà été appliqué sur le noeud c et n'a pu l'élaguer, il est inutile de l'appliquer à nouveau. Ou encore, lorsque le dernier fils du noeud c est élagué par le filtrage fort, on peut éviter de l'appliquer sur le noeud c et simplement backtracker comme si le filtrage fort avait élagué c.

```
procedure Divide&Conquer-BackwardPruning(c,level)
1     node[level] := c
2     credit := credit + C1
3     if credit < C2
4         if prune1(c) then return level-1
5     else
6         credit := credit - C2
7         if prune1(c) then return findLevel(dknp+1,level-1)
8         if prune2(c) then return findLevel(dknp+1,level-1)
9         else dknp := level
10    for s ∈ children(c) do
11        returnedLevel := BackwardPruning(s,level+1)
12        if returnedLevel < level return returnedLevel
13        else dknp := min(level, dknp)
14    return level-1

procedure findLevel(min,max)
1     if min > max then
2         dknp := min-1
3         return dknp
4     mid := floor((min+max)/2)
5     if prune2(node[mid]) then return findLevel(min,mid-1)
6     else return findLevel(mid+1,max)
```

FIG. 3 – Divide & Conquer Backward Pruning

trons que le coût du backward pruning est au plus $2 + \alpha$ fois le coût d'une exploration optimale, où α est le minimum entre la profondeur de l'arbre de recherche et le rapport C_2/C_1. Ensuite, nous en dérivons deux résultats importants en pratique :

1. Nous définissons une propriété d'incrémentalité *bottom-up* pour un algorithme de pruning, et nous montrons que pour un algorithme de pruning fort qui est incrémental, le coût du backward pruning est au plus à un facteur 3 de celui d'une exploration optimale.
2. Lorsque l'algorithme de pruning fort n'est pas incrémental, nous montrons que la version dichotomnique du backward pruning est nettement plus efficace que la version standard : son coût est au plus à un facteur $3 + \log_2(\alpha - 1)$ de celui d'une exploration optimale, où α est défini comme ci-dessus.

Les différents algorithmes de backtracking ne visitent pas exactement le même ensemble de noeuds : certains noeuds sont élagués par le filtrage fort mais pas par le filtrage faible. Par conséquent, suivant l'algorithme de filtrage utilisé en un noeud, ses sous-arbres peuvent être visités ou non. La comparaison des différents algorithmes de backtracking nécessite une analyse détaillée des relations entre ces algorithmes, cette analyse ainsi que les preuves des résultats théoriques sont données en annexe 1.

4.1 Résultats théoriques

Nous noterons BT_{bp} pour l'algorithme de backward pruning, et BT_{dcp} sa version dichotomique. Tout d'abord, la version standard du backward pruning peut être comparée au backtracking optimal :

Théorème 4.1 $cost(BT_{bp}) \leq (\alpha + 2)cost(BT_{opt})$*, où* α *est tel que :*
(1) $\alpha <$ *la profondeur maximale de l'arbre exploé par* BT_{bp}*; (2)* $\alpha \leq C_2/C_1$.

Le surcoût du backward pruning comparé à BT_{opt} vient des séquences de backtracks successifs dus au filtrage fort. A chaque backtrack, le filtrage fort est appelé ; ce comportement est responsable du facteur $(\alpha + 2)$ présent dans la borne de complexité du théorème 4.1 (les détails se trouvent dans la preuve du théorème 4.1). Pour certains algorithmes de filtrage forts, il est possible d'exploiter les séquences de backtracks successifs pour en améliorer le coût total. Nous définissons l'incrémentalité bottom-up comme suit :

Définition 4.2 *Soit* F *un noeud de l'arbre de recherche, et soit* C *un noeud fils de* F*. Un algorithme de filtrage fort* $prune_2$ *de coût* C_2 *est bottom-up incremental pour un algorithme de filtrage faible* $prune_1$ *de coût* C_1 *si et seulement si il peut être implémenté de telle façon que le coût de l'appel* $prune_2(C)$ *suivi de l'appel* $prune_2(F)$ *est au plus* $C_2 + C_1$ *(au lieu de* $2C_2$*).*

En corollaire, l'appel à $prune_2$ sur d noeuds d'une séquence de $d-1$ backtracks successifs coûte $C_2+(d-1)C_1$ et non dC_2. Par conséquent, lorsque le filtrage fort peut être rendu bottom-up incremental, on obtient une bien meilleure borne que celle du théorème 4.1 :

Théorème 4.3 *Lorsque* P_2 *est bottom up incremental :* $cost(BT_{bp}) \leq 3cost(BT_{opt})$

Si l'algorithme de filtrage fort n'est pas bottom-up incremental, il peut être intéressant d'utiliser la version dichotomique du backward pruning, comme le suggère le théorème suivant :

Théorème 4.4 $cost(BT_{dcp}) \leq (3 + \log_2(\alpha - 1))cost(BT_{opt})$*, où* α *est tel que :*
(1) $\alpha \leq$ *la profondeur maximale de l'arbre exploré par* BT_{dcp}*, (2)* $\alpha \leq C_2/C_1$

Ces théorèmes ainsi que la proposition 2.2 nous permettent de comparer le backward pruning aux algorithmes de backtracking BT_1 et BT_2 :

Corollaire 4.5
(i) Backward pruning peut gagner un facteur exponentiel en la profondeur de l'arbre comparé à BT_1*, et un facteur* C_2/C_1 *comparé à* BT_2.
(ii) Backward pruning (la version dichotomique) coûte au plus $(3 + \log_2(\alpha - 1))$ *fois* BT_1 *ou* BT_2*, où* α *est défini au théorème 4.4.*
(iii) De plus, si P_2 *est bottom up incremental : Backward pruning coûte au plus 3 fois* BT_1 *ou* BT_2.

5 Résultats expérimentaux

Nous comparons les différents algorithmes sur une famille d'arbres de recherche binaires qui simulent des résolutions de problèmes combinatoires typiques, où généralement les arbres ne sont pas équilibrés, les feuilles des arbres apparaissant à différentes profondeurs.

Notre modèle d'arbre de recherche définit la forme de l'arbre en fonction de 4 paramètres : d_{max}, la profondeur maximale de l'arbre, une profondeur d_0, avec $0 < d_0 \leq d_{max}$, et deux probabilités p_1 et p_2. Un arbre de recherche binaire est généré de telle façon que, à la profondeur d, l'opérateur de pruning P_i ait une probabilité $p_i(d)$ d'élaguer un noeud, où $p_i(d)$ est défini par deux simples interpolations linéaires (voir la figure 4) :

- à la profondeur 0, $p_i(0) = 0$,
- à la profondeur d_0, $p_i(d_0) = p_i$,
- à la profondeur d_{max}, $p_i(d_{max}) = 1$,
- à la profondeur d, $0 < d < d_0$, $p_i(d) = p_i * d/d_0$,
- à la profondeur d, $d_0 < d < d_{max}$, $p_i(d) = \frac{1*(d-d_0)+p_i*(d_{max}-d)}{(d_{max}-d_0)}$.

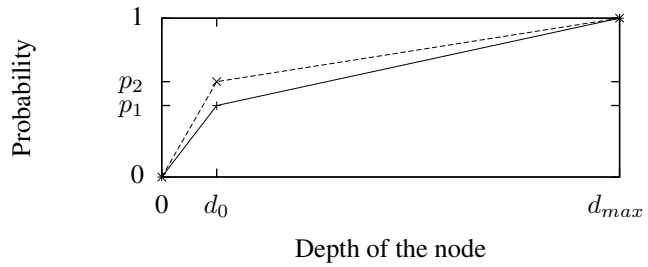


FIG. 4 – Probabilités d'élaguer un noeud en fonction de la profondeur du noeud

Pour comparer les différents algorithmes, nous normalisons leur coûts pour explorer l'arbre en les divisant par le coût du backtracking optimal. Par exemple, un rapport de 5 signifie que l'algorithme est 5 fois plus lent que l'algorithme le meilleur possible.

Les résultats sont présentés pour $d_{max} = 150$, $d_0 = 33$ et $p_1 = 0.45$. Ces valeurs ont été choisies pour que les arbres générés soient d'une taille raisonnable. Nous faisons varier p_2 de 0.45 à 1 par incrément de 0.05, puisque nous avons $p_2 \geq p_1$ (P_2 effectue un filtrage plus fort que P_1). Remarquons que, lorsque $p_2 = p_1 = 0.45$, P_2 ne peut élaguer aucun noeud qui n'est pas élagué par P_1. Lorsque $p_2 = 1$, P_2 élague toutes les branches de l'arbre à une profondeur au plus de d_0.

Le coût du filtrage P_1, C_1 est pris égal à 1, et les expérimentations ont été faites pour trois valeurs du coût du filtrage P_2 : $C_2 = 10, 100, 1000$. La figure 5 pésente les résultats pour la version dichotomique du backward pruning (BT_{dcp}), le backward pruning (BT_{bp}), le backward pruning avec un opérateur de pruning incremental ($Inc.BT_{bp}$), BT_1 et BT_2. Un point d'une des courbes de la figure 5 est une moyenne géométrique pour dix arbres de recherche générés aléatoirement. Les axes des ordonnées représentent les coût normalisés en échelle logarithmique, les axes des abscisses étant les valeurs de la probabilité p_2.

Les résultats sont en accord avec l'analyse théorique. Notons que $Inc.BT_{bp}$ a un bon comportement expérimental avec un coût d'environ 2 fois celui du backtracking optimal (la borne donnée par le théorème 4.3 est d'un facteur 3).

6 Discussion

6.1 Travaux connexes

De nombreux travaux destinés à améliorer l'algorithme de backtracking on été proposés, en particulier dans un contexte où il s'agit d'affecter des valeurs à un ensemble de variables soumises à des contraintes. Ces travaux peuvent généralement se classer en deux catégories (voir [3]) : les approches *look-ahead* et les approches *look-back*. Les approches *look-ahead*, qui exploitent des informations sur les variables non encore affectées, sont au coeur des solveurs SAT (qui utilisent des variations de l'algorithme DPLL [4, 5]), des solveur MIP (qui appliquent la programmation linéaire sur des relaxations continues du problème [6]), et des solveurs CP (qui utilisent diverses techniques de propagation [7]). Les approches *look-back* exploitent des informations sur la recherche déjà effectuée, en analysant le graphe de contraintes afin de déterminer de meilleur points de backtrack, ou d'apprendre de nouvelles contraintes entre les variables [8]. Notons aussi [9], qui présente un algorithme pour calculer un conflit minimal. Le backward pruning peut être considéré comme une nouvelle forme de méthode *look-back* dans laquelle l'analyse du graphe de contraintes est remplacée par l'exécution d'un algorithme au backtrack, un filtrage fort dans notre cas. De plus, le backward pruning, n'étant pas restreint à l'effectation de valeurs à des variables, est un peu plus général.

Le travail le plus proche du backward pruning est Quickshaving [10]. A chaque backtrack, Quickshaving vérifie les précédentes décisions de recherche qui ont conduit à un échec. On peut voir ces vérifications additionnelles comme une forme de filtrage fort. QuickShaving est une amélioration du backtracking dans la mesure où il peut être exponentiellement plus rapide au prix d'un faible surcoût. Cependant, contrairement au backward pruning, QuickShaving n'est pas toujours une amélioration du backtracking avec un filtrage fort et peut être exponentiellement plus lent. En fait, QuickShaving n'a pas l'équivalent des propriétés d'optimalité du backward pruning.

Différentes approches pour combiner un filtrage faible

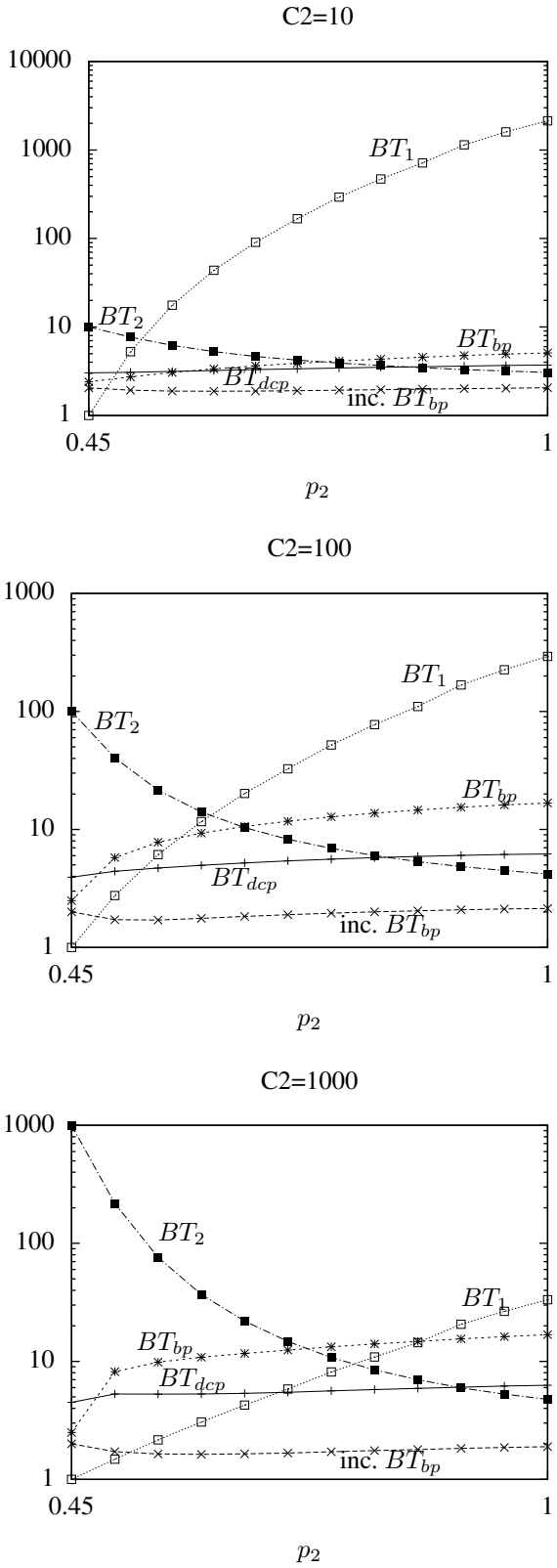


FIG. 5 – Coûts normalisés (axe des y en échelle logarithmique) pour p_2 variant de p_1 à 1.

et un filtrage fort ont été proposées dans la littérature. En MIP, le filtrage fort prend la forme d'un ajout de contraintes additionelles qui doivent être vérifiées par toute solution entière (*cut*). Générer de tels cuts est coûteux, et, en pratique, des règles heuristiques permettent de décider si un nouveau cut doit être gardé ou non. En CP, mentionnons d'abord [11] : deux techniques de pruning pour les réseaux de contraintes binaires sont analysées afin de déterminer des cas où elles donnent toutes deux le même résultat ; pendant la recherche, si l'on s'aperçoit que l'on se trouve dans un de ces cas, le pruning faible est appliqué, autrement, le pruning fort est appliqué. Dans [12], pour les problèmes continus, l'appel au pruning fort est évité lorsqu'une méthode d'extrapolation, peu coûteuse, prédit que le filtrage fort sera inutile. Finalement, [13] et [14] proposent des heuristiques spécifiques pour choisir entre deux techniques de pruning. Aucune de ces approches ne garantit de propriété d'optimalité proche de celles de backward pruning.

Les résultats de complexité donnés dans ce papier sont une forme de *competitive analysis*. La *competitive analysis* a été introduite par [15] pour analyser des problèmes online, c'est-à-dire des problèmes dans lesquels il faut satisfaire une séquence de requêtes sans connaître les futures requêtes. La performance d'un algorithme online est comparée à un algorithme offline, qui lui connaît à l'avance la séquence de requêtes. En fait, le backtracking optimal, comme défini en section 2.2, peut se voir comme un algorithme offline : il connaît à l'avance le meilleur pruning à appliquer à chaque noeud. Dans le vocabulaire de la *competitive analysis*, nos résultats de complexité (proposition 2.2, théorèmes 4.1, 4.3 et 4.4) sont la détermination des *competitive ratios* des différents algorithmes de backtracking.

6.2 Perspectives

Dans ce papier, nous avons supposé que deux prédicats de pruning étaient donnés, l'un plus fort et plus coûteux que l'autre. Discutons de ces hypothèses :

- Si les deux prédicats ont un coût similaire, il n'y a aucune raison d'utiliser le backward pruning, le backtracking avec le prédicat de pruning fort est optimal ou proche de l'optimal.
- Lorsque les puissances d'élagage des deux prédicats ne peuvent pas être comparées, c'est-à-dire lorsque aucun n'est plus fort que l'autre, le framework du backward pruning peut encore s'appliquer. Appelons $prune_2$ le plus coûteux des deux, nous pouvons définir $prune'_2$ comme suit : d'abord, appliquer $prune_1$, et ensuite appliquer $prune_2$. Ensuite, nous considérons les deux prédicats de pruning $prune_1$ et $prune'_2$.
- Considérons le cas d'un seul algorithme de pruning : dans ce cas, la question à chaque noeud se ramène à décider si l'algorithme de pruning est appliqué ou

non avant l'expansion du noeud. Le backward pruning peut s'appliquer en définissant $prune_1$ qui renvoie toujours faux, $prune_2$ étant l'algorithme de pruning donné.
– Nous pouvons aussi considérer plus de deux algorithmes de pruning. Ce cas peut se traiter par une petite extension du backward pruning.
– Ce papier s'est focalisé sur le cas général du backtracking où l'on ne sait rien du problème. Souvent, le problème est d'affecter des valeurs à des variables. Dans ce cas, au lieu d'un prédicat de pruning, on peut faire appel à une procédure de pruning *look ahead*, qui peut élaguer des descendants. Le principe du backward pruning fonctionne encore, cependant certaines adaptations des algorithmes et des résultats de complexité sont nécessaires, mais sont hors du cadre de ce papier.

Le backward pruning étant un schéma générique, une perspective générale est d'étudier spécifiquement différent domaines d'application. Par exemple, les théorèmes 4.1, 4.3 et 4.4 sont vrais lorsque les temps d'exécution des algorithmes de pruning sont constants. Lorsque ce n'est pas le cas, ces résultats peuvent encore être utilisés à condition que l'on dispose de bornes sur les temps d'exécution. Pour un problème donné, cependant, le temps d'exécution des algorithmes peut fortement varier, et une étude théorique spécifique peut être nécessaire pour obtenir des bornes précises sur le temps d'exécution du backward pruning, ainsi que pour spécialiser l'idée du backward pruning pour ce problème. Par exemple, nous avons intégré une variante du backward pruning dans la méthode de recherche par defaut d'un solveur de contraintes industriel (CP Optimizer dans IBM ILOG CPLEX Optimization Studio [2]). Dans un solveur de contraintes, les temps d'exécution des algorithmes de pruning sont très variables, ce qui a demandé quelques adaptations du schéma théorique décrit dans ce papier.

7 Conclusions

Ce papier apporte quelques éléments de réponse à la question générale de l'amélioration des algorithmes de recherche combinatoire. De nombreux solveurs combinatoires modernes, s'ils tirent leur efficacité d'une interaction complexe de plusieurs composants, reposent fortement sur l'algorithme de backtracking. Nous avons introduit le backward pruning, un algorithme de backtracking qui exploite l'idée contre-intuitive d'élaguer l'arbre de recherche en partant des feuilles de l'arbre. Nous montrons, théoriquement et expérimentalement, que le backward pruning améliore les performances pour une large classe de problèmes, ce qui laisse supposer qu'il pourrait remplacer avec succès le backtracking dans un certain nombre d'applications.

De plus, grâce à la borne théorique donnée par le "backtracking optimal", nous savons que les algorithmes de backward pruning ne sont pas très loin de la meilleure façon possible d'explorer un arbre de recherche.

Remerciements

Je remercie Jean-François Puget, Philippe Refalo et Paul Shaw pour leurs relectures et commentaires constructifs.

Annexe 1 : preuves et analyse détaillée

Proof. [of Proposition 2.2] Backtracking is used to solve a problem characterized by its size n. Typically, this size corresponds to the maximal depth of the search tree generated by backtracking. The maximal saving of BT_{opt} compared to BT_1 happens when the optimal exploration consists of applying P_2 to the root node, which proves that no solution exists, whereas BT_1 needs to develop a tree with branching factor $r \geq 2$ and of depth n : BT_{opt} costs C_2 and BT_1 costs $C_1 r^n$. Therefore, as C_1 and C_2 are constant, we have (i).

The gain of BT_{opt} compared with BT_2 is maximal when P_2 does not prune more nodes than P_1 : in this case, BT_{opt} is equivalent to BT_1 and is C_2/C_1 times more efficient than BT_2. Thus, we have (ii).

Finally, we can see that BT_{opt} may save an exponential running time compared to BT_1 and a factor C_2/C_1 in running time compared to BT_2 on the same problem, by considering a search tree where the root node has two children A and B such that : (1) for exploring A, BT_1 needs to visit 2^n nodes, and BT_2 needs to visit 1 node ; (2) for exploring B, BT_1 and BT_2 need to visit M nodes, where M is a big number compared to C_2/C_1, but is small compared to 2^n : $C_2/C_1 \ll M \ll 2^n$. The running times of BT_1, BT_2 and BT_{opt} are respectively equal to $C_1(2^n + M) \approx 2^n C_1$, $C_2(1 + M) \approx MC_2$ and $C_2 + C_1 M \approx MC_1$. Thus, we have (iii). □

In order to prove Theorems 4.1, 4.3 and 4.4, we need to introduce some additional definitions and lemmas. Let us call $Tree_1$, $Tree_2$, $Tree_{opt}$, $Tree_{bp}$ and $Tree_{dcp}$ the search trees respectively generated by BT_1, BT_2, BT_{opt}, BT_{bp} and BT_{dcp}.

Définition 7.1 *We can associate a label to each node N of a search tree, which denotes the unique path from the root to N specified by the rank of every child along the path :* $label(N) = (r_0, r_1, \ldots, r_d)$.

For instance, a label $(2, 1, 4)$ corresponds to the fourth child of the first child of the second child of the root.

Then, we define the concept of *position* of a backtracking search in a search tree. As a backtracking algorithm may backtrack several times to the same node, the label of

a node alone is not enough to specify exactly the position in a search tree.

Définition 7.2 *A position in a search tree is denoted by a couple (N, k), where N is a node of the search tree, and k is the number of children of N that have been already visited.*

We will need to compare two positions in a tree, possibly in two different trees (generated by different backtrack algorithms for the same problem). First, we extend the concept of label to position :

Définition 7.3 *Let $A = (N_A, k)$ be a position, let $label(N_A) = (r_0, r_1, \ldots, r_d)$, then $extendedLabel(A) = (r_0, r_1, \ldots, r_d, k, +\infty)$*

Now, we can define an order on positions, which simply corresponds to the order in which the positions are visited with a left to right depth first traversal of the tree (i.e., a backtracking algorithm) :

Définition 7.4 *Let A and B be two positions in a search tree. We say that A is smaller that B, and we note $A < B$, if and only if $extendedLabel(A)$ is lexicographically smaller than $extendedLabel(B)$.*

We will need to have a measure of the cost spent by a backtracking algorithm to go from a position to another one. This will be possible thanks to the following *distance*, which expresses the number of nodes opened by a left to right depth first traversal of the tree between the two positions (in other words the number of recursive calls to the backtracking procedure). Note that a node N is opened at position $(N, 0)$.

Définition 7.5 *Let A and B be two positions in a search tree such that $A \leq B$, $distance(A, B, Tree)$ denotes the number of positions C in the tree $Tree$ such that : $C > A$, $C \leq B$ and $C = (*, 0)$.*

Example : let $A = (N, k)$ be a position such that k is equal to the total number of children of node N. That is to say that all the children of node N have been just explored. Let $B = (F, i)$ be a position such that N is the i-th child of node F. In other words, B is the position just after A, and is reached after a simple backtrack from A. Then, by definition of distance, and since $i \neq 0$, we have : $distance(A, B, Tree) = 0$. That is to say that the distance when backtracking is 0. Then, consider the position $C = (N', 0)$ where N' is the $i + 1$-th child of F. We have $distance(A, C, Tree) = distance(B, C, Tree) = 1$.

Définition 7.6 *Let A and B be two positions in the search tree generated by backtracking algorithm BT such that $A \leq B$, $cost(A, B, BT)$ denotes the cost spent by BT for going from position A to position B.*

Note that, as BT_1 applies P_1 to each node opened between A and B, $cost(A, B, BT_1) = C_1 * distance(A, B, Tree_1)$.

All the nodes which are explored by BT_{opt} are also explored by BT_1. Furthermore, when a node is expanded into its children, the children are the same in BT_{opt}, BT_{bp}, BT_{dcp}, BT_1 or BT_2. Therefore, as a position is charaterized by a node and the rank of one of its children, a position in the tree $Tree_{opt}$ always exists in the tree $Tree_1$. Obviously, the converse is not true : a position in $Tree_1$ may not exist in $Tree_{opt}$.

The following lemma is needed for the proof of Theorem 4.1 :

Lemme 7.7 *Let $A = (N_A, *)$ and $B = (N_B, *)$ be two positions in $Tree_1$, and such that $A < B$. Assume that $P_2(N_A) \neq \emptyset$ and $P_2(N_B) \neq \emptyset$ (neither N_A nor N_B can be pruned by P_2), Then, we have :*

(i) A and B also exist in $Tree_{opt}$,

(ii) if $distance(A, B, Tree_1) \geq C_2/C_1$, then, $cost(A, B, BT_{opt}) \geq C_2$,

(iii) if $distance(A, B, Tree_1) < C_2/C_1$, then, $cost(A, B, BT_{opt}) = cost(A, B, BT_1) = distance(A, B, Tree_1) * C_1$.

Proof. P_2 cannot prune N_A nor N_B. By monotonicity of P_2, no ancestor of N_A or N_B can be pruned by P_2. Thus, we know that positions A and B also exist in the search tree generated by BT_{opt}, which proves (i). Furthermore, BT_{opt} does not apply P_2 on any ancestor of N_A or N_B, otherwise it would not be optimal. Let us consider two cases.
Case (1) : BT_{opt} applies P_2 at a position such that this pruning changes the distance : $distance(A, B, Tree_{opt}) < distance(A, B, Tree_1)$. Then, as no ancestor of N_A or N_B can be pruned by P_2, this position must be strictly between the positions A and B, and, thus, the cost for BT_{opt} to go from A to B is at least C_2.
Case (2) : BT_{opt} does not apply P_2 at a position such that this pruning changes the distance, thus, $distance(A, B, Tree_{opt}) = distance(A, B, Tree_1)$. As BT_{opt} spend at least C_1 at each open node, we have $cost(A, B, BT_{opt}) \geq distance(A, B, Tree_{opt}) * C_1 = cost(A, B, BT_1)$. By optimality, BT_{opt} cannot cost more than BT_1, then, it costs exactly $cost(A, B, BT_1)$.

Let us prove point (ii) of the lemma. We have $distance(A, B, Tree_1) \geq C_2/C_1$. Thus $distance(A, B, Tree_1) * C_1 \geq C_2$, thus, both in case (1) and case (2) we have $cost(A, B, Tree_{opt}) \geq C_2$, which proves (ii).

Now, let us prove (iii) : we have $distance(A, B, Tree_1) < C_2/C_1$. Then, $cost(A, B, BT_1) < C_2$. By optimality of BT_{opt}, we have $cost(A, B, BT_{opt}) \leq cost(A, B, BT_1)$. Thus, BT_{opt} cannot apply P_2 between A and B because this would cost at least C_2, therefore case (1) is not possible. Thus, we

are in case (2) and $cost(A, B, BT_{opt}) = cost(A, B, BT_1)$ which proves (iii). □

We now formalize BT_{bp} as a sequence of positions :

Définition 7.8 *For simplifying the formalization, we consider that the root node has a father, called top, which has only one child, the root node. Furthermore, let us assume that no pruning operator can prune top : $P_2(top) \neq \emptyset$ and $P_1(top) \neq \emptyset$. This additional node amounts to add two fake positions, position $start = (top, 0)$ and position $end = (top, 1)$, as first and last element to the sequence of real positions visited by a backtracking algorithm. Let $S_0, \ldots, S_w$ be the sequence of positions visited by BT_{bp} for exploring the whole tree starting at $start$ and ending at end. We have : $S_0 = start, S_w = end$. Note that, consistently with our definition of distance, we have $distance(start, S_1, Tree_{bp}) = 1$ and $distance(S_{w-1}, end, Tree_{bp}) = 0$.*

We define $Seq_{bp} = \{A_i\}_{i=0\ldots v}$ a subsequence of the above sequence $\{S_i\}_{i=0\ldots w}$ where $A_0 = S_0 = start$, $A_v = S_w = end$, and such that for $i \in [1, v-1]$, A_i is the first position at a distance of at least $k = C_2/C_1$ after A_{i-1} which cannot be pruned by P_2. Formally : $A_i = (N_{A_i}, *)$ is the smallest position in $\{S_i\}_{i=0\ldots w}$ such that $distance(A_{i-1}, A_i, Tree_{bp}) \geq k$ and $P_2(N_{A_i}) \neq \emptyset$.

This subsequence $Seq_{bp} = \{A_i\}_{i=0\ldots v}$ formalizes the behavior of BT_{bp} in the following way : let us start from a position A_i of the subsequence, where $i \in [0, v-1]$, and assume the global variable *credit* equals 0 when the algorithm is at this position. After $k = C_2/C_1$ applications of P_1, and k increases of *credit* by C_1, *credit* reaches the value C_2. Then, BT_{bp} reset *credit* to 0 and, possibly after a backtrack due to a pruning from P_1, applies P_2 on a position called B. If P_2 cannot prune the node at position B, we take $A_{i+1} = B$. If P_2 can prune the node at position B, BT_{bp} backtracks, reapplies P_2 on the father of B, and continues to backtrack until P_2 can no longer prune the current position. We call A_{i+1} this position.

The last position A_v may be different : it handles the case when the credit does not reach the value C_2 and BT_{bp} ends.

We are now able to prove Theorems 4.1, 4.3, and 4.4.

Proof. [of Theorem 4.1] Let $Seq_{bp} = \{A_i\}_{i=0\ldots v}$, let $k = C_2/C_1$. We first compute the cost of BT_{bp} for going from A_i to A_{i+1}, for $i \in [0, v-2]$, and, then, the cost of BT_{bp} for going from A_{v-1} to A_v :

1. The cost of BT_{bp} for going from A_i to A_{i+1}, for $i \in [0, v-2]$ is :
   $cost(A_i, A_{i+1}, BT_{bp}) = k * C_1 + (n^i_{bp} + 1) * C_2 = (n^i_{bp} + 2) * C_2$, where n^i_{bp} is the number of backtracks due to P_2 to find the position A_{i+1}, i.e., the number of backward prunings.
   Furthermore, as $distance(A_i, A_{i+1}, Tree_{bp}) \geq k$, we also have $distance(A_i, A_{i+1}, Tree_1) \geq k$. Thus, thanks to lemma 7.7 (ii) $cost(A_i, A_{i+1}, BT_{opt}) \geq C_2$. Thus we have : $cost(A_i, A_{i+1}, BT_{bp}) \leq (n^i_{bp} + 2) * cost(A_i, A_{i+1}, BT_{opt})$.
2. It remains to compute the cost for going from A_{v-1} to A_v. We consider two cases :
   (a) Either we have $distance(A_{v-1}, A_v, Tree_{bp}) \geq k$, thus, P_2 is applied on A_v, and the same argument as above applies, we have : $cost(A_{v-1}, A_v, BT_{bp}) \leq (n^{v-1}_{bp} + 2) * cost(A_{v-1}, A_v, BT_{opt})$.
   (b) Or we have $distance(A_{v-1}, A_v, Tree_{bp}) < k$. As credit is equal to 0 at position A_{v-1}, credit does not reach value C_2 and P_2 is not applied between A_{v-1} and A_v. Thus, backward pruning is equivalent to BT_1 between A_{v-1} and A_v : $cost(A_{v-1}, A_v, BT_{bp}) = cost(A_{v-1}, A_v, BT_1)$. Furthermore, A_v is the last position visited by BT_{bp}, i.e., $A_v = (top, 1)$, and we know that P_2 cannot prune the node top. Thus, we can apply lemma 7.7 (iii), from which we get : $cost(A_{v-1}, A_v, BT_{opt}) = cost(A_{v-1}, A_v, BT_1)$. Thus, $cost(A_{v-1}, A_v, BT_{bp}) = cost(A_{v-1}, A_v, BT_{opt})$.

Finally, by summing the costs for each part A_i, A_{i+1}, and by taking α equal to the maximal number of backtracks due to P_2, we get : $cost(BT_{bp}) \leq (\alpha + 2) cost(BT_{opt})$, where $\alpha = max_{i \in [0, v-1]} n^i_{bp}$.

Clearly $\alpha < depth(Tree_{bp})$, which proves the first bound of the theorem. The second bound of the theorem comes from $\alpha \leq C_2/C_1$. This can be seen by considering the position B where BT_{bp} applies P_2 for the first time after A_i. Position B is at distance C_2/C_1 from A_i. BT_{bp} does not backtrack higher than the most recent common ancestor of N_{A_i} and N_B (by monotonicity of P_2 since this is an ancestor of N_{A_i}, and thus P_2 cannot prune this ancestor). The number of ancestors of N_B, including B itself, that are not ancestor of N_{A_i} is at most C_2/C_1 since $distance(A_i, B, Tree_{bp}) = C_2/C_1$. □

Proof. [of Theorem 4.3] We do not give the complete proof, it is quite similar to the previous one. Instead, we simply recompute the cost of BT_{bp} for going from A_i to A_{i+1}. As P_2 is bottom up incremental, the cost for applying $n^i_{bp} + 1$ times P_2 while backtracking is equal to $C_2 + n^i_{bp} * C_1$.

Thus, the cost of BT_{bp} for going from A_i to A_{i+1} is : $cost(A_i, A_{i+1}, BT_{bp}) = k * C_1 + C_2 + n^i_{bp} * C_1 = (n^i_{bp}) * C_1 + 2 * C_2$.

As we have seen in the previous proof, $n^i_{bp} \leq C_2/C_1$. Thus, $cost(A_i, A_{i+1}, BT_{bp}) \leq 3 * C_2$.

By summing, we get $cost(BT_{bp}) \leq 3 * cost(BT_{opt})$. □

Proof. [of Theorem 4.4] The point is to note that BT_{dcp} can also be characterized by the sequence $Seq_{bp} = \{A_i\}_{i=0...v}$. The only difference with BT_{bp} is that, for reaching position A_{i+1} starting from A_i, BT_{bp} needs to apply at most $\lfloor \log_2(d-1) \rfloor + 2$ times pruning operator P_2, where d is the depth of the search tree (we assume $d > 1$, otherwise the theorem is trivially true). The cost of BT_{dcp} for going from A_i to A_{i+1} is : $cost(A_i, A_{i+1}, BT_{dcp}) \leq kC_1 + C_2 \lfloor \log_2(d-1) + 2 \rfloor = C_2(3 + \lfloor \log_2(d-1) \rfloor)$. Indeed, BT_{dcp} calls the divide-and-conquer search between *dknp*+1 and *level*. Furthermore, we have that *level-dknp* $\leq C_2/C_1$. Thus, $cost(A_i, A_{i+1}, BT_{dcp}) \leq C_2 * (3 + \lfloor \log_2(C_2/C_1 - 1) \rfloor)$. By summing, we prove the theorem. □

Références

[1] Golomb, S.W., Baumert, L.D. : Backtrack programming. J. ACM **12** (1965) 516–524

[2] IBM : IBM ILOG CPLEX Optimiation Studio 12.2 User's manual. IBM (2010)

[3] Dechter, R. : Enhancement schemes for constraint processing : Backjumping, learning, and cutset decomposition. Artificial Intelligence **41** (1990) 273–312

[4] Davis, M., Putnam, H. : A computing procedure for quantification theory. J. ACM **7** (1960) 201–215

[5] Davis, M., Logemann, G., Loveland, D. : A machine program for theorem-proving. Commun. ACM **5** (1962) 394–397

[6] Wolsey, L.A. : Integer programming. Wiley (1998)

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

[8] Stallman, R.M., Sussman, G.J. : Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. Artif. Intell. **9** (1977) 135–196

[9] Junker, U. : Quickxplain : preferred explanations and relaxations for over-constrained problems. In : AAAI'04 : Proceedings of the 19th national conference on Artifical intelligence, AAAI Press (2004) 167–172

[10] Lhomme, O. : Quick shaving. In Veloso, M.M., Kambhampati, S., eds. : AAAI, AAAI Press / The MIT Press (2005) 411–415

[11] Sakkout, H.E., Wallace, M., Richards, B. : An instance of adaptive constraint propagation. In Freuder, E.C., ed. : CP. Volume 1118 of Lecture Notes in Computer Science., Springer (1996) 164–178

[12] Lebbah, Y., Lhomme, O. : Accelerating filtering techniques for numeric csps. Artif. Intell. **139** (2002) 109–132

[13] Stergiou, K. : Heuristics for dynamically adapting propagation. In Ghallab, M., Spyropoulos, C.D., Fakotakis, N., Avouris, N.M., eds. : ECAI. Volume 178 of Frontiers in Artificial Intelligence and Applications., IOS Press (2008) 485–489

[14] Szymanek, R., Lecoutre, C. : Constraint-level advice for shaving. In de la Banda, M.G., Pontelli, E., eds. : ICLP. Volume 5366 of Lecture Notes in Computer Science., Springer (2008) 636–650

[15] Sleator, D.D., Tarjan, R.E. : Amortized efficiency of list update and paging rules. Commun. ACM **28** (1985) 202–208

Satisfiabilité minimale et applications *

Chu-Min Li [1,2], **Zhu Zhu**[2], **Felip Manyà**[3], **Laurent Simon**[4]

[1]Huazhong Univ. of Science and Technology, Wuhan, China chu-min.li@u-picardie.fr
[2]Univ. de Picardie Jules Verne Amiens, France zhu.zhu@u-picardie.fr
[3]IIIA-CSIC Bellaterra, Spain felip@iiia.csic.es
[4]Univ. Orsay Paris Sud 11 Orsay, France simon@lri.fr

Abstract

Nous proposons, dans cet article, de placer le problème de satisfiabilité minimale (MinSAT) sur le devant de la scène. Pour cela, un algorithme efficace de type *branch-and-bound* pour résoudre le problème MinSAT partiel avec pondération est introduit, permettant du même coup une évaluation empirique de cet algorithme pour les cas Min-3SAT aléatoires, MaxClique et pour les problèmes d'enchères combinatoires. Les techniques employées pour résoudre MinSAT se démarquent de celles utilisées pour le problème de satisfiabilité maximale (MaxSAT), qui a lui été très étudié. Nos résultats démontrent de manière empirique que, notamment sur les problèmes d'enchères combinatoires, une réduction du problème initial à MinSAT peut être plus intéressante qu'une réduction à MaxSAT, ou même que l'utilisation de méthodes dédiées. Nous étendons par ailleurs ce travail en montrant une corrélation entre le nombre minimal et le nombre maximal de clauses satisfaites dans une instance SAT aléatoire donnée.

1 Introduction

La résolution pratique de problèmes NP-Complets, via une réduction au problème SAT de la logique propositionnelle, est une technique puissante qui a permis, et permet encore, de résoudre des problèmes académiques ainsi que des problèmes d'importance industrielle. Plus récemment, les succès obtenus autour de SAT ont permis d'envisager des formalismes liés, comme MaxSAT, MaxSAT partiel ou encore MaxSAT partiel pondéré [Li and Manyà 2009]. Les résultats obtenus, plus que prometteurs, montrent combien ils deviennent des approches génériques crédibles pour résoudre, en pratique, des problèmes d'optimisations, à l'instar de SAT et des problèmes de décision.

Dans cet article, nous proposons de prendre une nouvelle direction. Plutôt que de se focaliser sur le formalisme MaxSAT, qui tente de maximiser le coût des clauses satisfaites, nous allons nous concentrer sur le problème opposé, c'est à dire tenter de minimiser le coût des clauses satisfaites. De manière plus précise, nous nous focalisons sur le problème MinSAT partiel pondéré, dans lequel les instances sont représentées par des ensembles de clauses, chaque clause étant déclarée dure (*hard*) ou molle (*soft*), et ayant éventuellement un poids particulier. Résoudre le problème MinSAT partiel pondéré revient à trouver une assignation des variables qui satisfasse toutes les clauses dures, et qui minimise la somme des poids des clauses molles. Ce problème de minimisation est intéressant pour, notamment, deux raisons : (1) Certains problèmes admettent un encodage plus naturel en les considérant comme des problèmes de minimisation plutôt que des problèmes de maximisation, (2) Chercher à minimiser va nous permettre d'introduire une nouvelle technique, relativement puissante, de bornage supérieur, qui ne peut être appliquée aux solveurs MaxSAT basés sur les techniques branch-and-bound. Alors que les deux problèmes MaxSat et MinSat sont tous deux des extensions naturelles de SAT, les techniques que l'on peut mettre en œuvre pour les résoudre sont bien différentes en pratique.

Plus concrètement, nous proposons dans la suite de cet article un algorithme de type branch-and-bound pour le problème MinSAT pondéré partiel, intégrant de nouvelles techniques de bornage. Cet algorithme, appelé $MinSatz$ est également testé grâce à une importante étude empirique. Celle-ci prend en compte les problèmes aléatoires Min-3SAT, MaxClique ainsi que les problèmes d'enchères combinatoires. Les résultats obtenus démontrent combien une réduction à MinSAT peut être plus efficace qu'une réduction à MaxSAT sur ces problèmes, y compris en com-

*Ce travail est soutenu par le projet français ANR UNLOC (ANR-08-BLAN-0289-03), par les projets espagnols Consolider-Ingenio10 CSD2007-0022, TIN2010-20967-C04-01, SGU P. N. de Movilidad de Recursos Humanos, ainsi que par les projets GenCat 2009-SGR-1434, et par le National Natural Science Foundation of China (NSFC) grant No. 61070235.

parant nos résultats avec d'autres méthodes dédiées du domaine. L'étude du problème MinSAT nous permet aussi de publier ici une corrélation intrigante entre la valeur MinSAT et la valeur MaxSAT sur les problèmes 3SAT.

À notre connaissance, cet article introduit le premier solveur exact spécifique pour les instances MinSAT partielles pondérées (et les variantes MinSAT, MinSAT pondérées et MinSAT partielles). C'est aussi la première fois que le formalisme MinSAT est utilisé, avec succès, pour résoudre des problèmes concrets d'optimisation. Ce travail a été précédé de, notamment, [Li *et al.* 2010b], dans lequel les auteurs proposaient différents moyens d'encoder le problème MinSAT (non pondéré) en un problème MaxSAT. Cette réduction ne permettait pas une généralisation aux instances MinSAT partielles pondérées qui nous intéressent ici. Par ailleurs, les expérimentations étaient limitées aux problèmes aléatoires. Notons aussi que, même s'il n'avait jusqu'à présent été étudié que de manière parcellaire (par rapport au problème MaxSAT), le problème MinSAT (non pondéré) a tout de même été étudié dans quelques travaux (voir par exemple [Avidor and Zwick 2005, Kohli *et al.* 1994, Marathe and Ravi 1996] et les références incluses).

La suite de cet article est structuré de la manière suivante : la section 2 contient les définitions de base en graphe et autour du problème MinSAT. La section 3 décrit le solveur MinSAT que nous proposons, accompagné d'une description de ses techniques de bornage. La section 4 présente les problèmes et les solveurs utilisés dans l'évaluation empirique, ainsi qu'une discussion des résultats obtenus. Bien entendu, la section 5 conclut l'article en proposant notamment quelques directions de recherche possibles.

2 Préliminaires et notations

Un littéral est une variable propositionnelle ou sa négation. Une clause est une disjonction de littéraux. Une clause pondérée est une paire (c, w), où c est une clause et w, son poids, est un entier naturel ou infini. Une clause est dure si son poids est l'infini ; sinon est appelée molle. Une instance de MinSAT partiel pondéré (idem pour MaxSAT) est un multi-ensemble de clauses pondérées $\phi = \{(h_1, \infty), \ldots, (h_k, \infty), (c_1, w_1), \ldots, (c_m,\ w_m)\}$, où les premières k clauses sont dures et les dernières m clauses sont molles. Par soucis de simplicité, dans la suite, nous omettrons de spécifier les poids infinis, et écrirons plutôt $\phi = \{h_1, \ldots, h_k, (c_1, w_1), \ldots, (c_m, w_m)\}$.

Une interprétation est une assignation de chaque variable à 0 ou 1. Le coût d'une interprétation l de ϕ est la somme des coûts des clauses satisfaites par l. Le problème MinSAT partiel pondéré, pour une instance ϕ, revient à trouver une interprétation des variables de coût minimum qui satisfasse toutes les clauses dures (i.e. une assignation optimale), tandis que le problème MaxSAT partiel pondéré revient quant à lui à trouver l'assignation de coût maximum satisfaisant toutes les clauses dures. De manière exhaustive nous pouvons définir tous les sous-problèmes : le problème MinSAT (MaxSAT) partiel est le problème MinSAT (MaxSAT) partiel pondéré dans lequel toutes les clauses molles sont de poids 1. Le problème MinSAT (MaxSAT) est le problème MinSAT (MaxSAT) partiel dans lequel il n'y a pas de clauses dures. Le problème SAT est le problème MaxSAT partiel dans lequel il n'y a pas de clauses molles.

Une *clique* d'un graphe non dirigé $G = (V, E)$, où V est l'ensemble des noeuds et E l'ensemble des arcs, est un sous-ensemble C de V tel que, pour chaque paire de noeuds de C, il existe un arc qui les connecte. Cela revient à dire que le sous-graphe induit par C est complet. Une clique maximale est une clique de la plus grande taille possible. Le *problème de clique maximale (MaxClique)* pour un graphe non dirigé G revient à trouver une clique maximale dans G. Une *partition en cliques* d'un graphe non dirigé $G = (V, E)$ est une partition de V en ensembles disjoints $V_1, \ldots, V_k$ telle que, pour $1 \leq i \leq k$, le sous-graphe induit par V_i est un graphe complet. Soit $\chi(G)$ le nombre minimum de couleurs nécessaires pour colorier les noeuds de G de façon à ce que deux noeuds adjacents soient de couleurs différentes, et soit $\omega(G)$ la taille de la clique maximale de G, on dit que G est parfait si $\chi(G')$=$\omega(G')$ pour tout sous-graphe induit G' de G.

3 Un solveur MinSAT exact

Nous introduisons dans cette section le premier solveur spécifique pour les instances MinSAT partielles pondérées (à notre connaissance), appelé MinSatz, basé sur Satz [Li and Anbulagan 1997], qui a d'ailleurs aussi servi de base à MaxSatz [Li *et al.* 2007]. Pour des raisons de simplicité nous commencerons par présenter le cas MinSAT partiel (non pondéré). MinSatz se base sur le schéma classique branch-and-bound, dont l'espace de recherche est formé par un arbre représentant toutes les interprétations possibles. Contrairement à un solveur MaxSAT de type branch-and-bound, comme MaxSatz, qui résout les problèmes MaxSAT en minimisant le nombre de clauses non satisfaites, MinSatz doit quant à lui maximiser le nombre de clauses falsifiées.

A chaque nœud de l'arbre de recherche, MinSatz commence par appliquer la propagation unitaire sur les seules clauses dures (i.e. étant donnée une clause dure l, initiale ou nouvellement dérivée, MinSatz satisfait et retire toutes les clauses contenant le littéral l, et efface toutes les occurrences de $\neg l$; les clauses molles ne sont pas propagées dans la mesure où elles ne sont pas forcément satisfaites). Si l'une des clauses dures devient vide, MinSatz revient sur ses choix précédents. Sinon, il calcule une borne supérieur du nombre maximal des clauses molles falsifiées (UB) si l'assignation partielle courante est étendue jusqu'à devenir

complète. Ce nombre, UB, est comparé avec le nombre de clauses falsifiées, associé à la meilleur assignation découverte jusque là (LB). Si $UB \leq LB$, une meilleure solution ne peut pas être trouvée à partir du nœud courant, et MinSatz doit aussi revenir en arrière. Sinon, une variable est sélectionnée et instanciée. Tant que l'espace de recherche n'a pas été totalement exploré, MinSatz répète ce procédé. Lorsque la recherche est finie, MinSatz retourne la meilleur solution trouvée. L'algorithme 1 détaille le pseudo-code de MinSatz.

Algorithm 1: MinSatz(ϕ, LB)

$\phi \leftarrow$ hardUnitPropagation(ϕ) ;
Si ϕ contient une clause dure vide **Alors Retourner** -1 ;
Si ϕ={} **Ou** ϕ ne contient que des clauses molles vides **Alors Retourner** #empty(ϕ) ;
UB $\leftarrow$ #empty(ϕ)+overestimation(ϕ) ;
Si (UB $\leq$ LB) **Alors Retourner** LB ;
x $\leftarrow$ select(ϕ) ;
LB $\leftarrow$ MinSatz(ϕ_x, LB) ;
LB $\leftarrow$ MinSatz($\phi_{\neg x}$, LB) ;
Retourner LB.

Pour résoudre une instance ϕ, il faut appeler MinSatz(ϕ, 0). Si MinSatz retourne -1, cela signifie que la partie dure des clauses de ϕ est insatisfiable, et il n'y a pas de solution possible. La fonction #empty(ϕ) retourne le nombre de clauses molles vides dans ϕ ; overestimation(ϕ) retourne une surestimation du nombre de clauses molles qui seraient insatisfiables si l'assignation partielle courante était étendue jusqu'à être complète. select(ϕ) retourne la variable la plus fréquente dans ϕ. L'instance ϕ_x (resp. $\phi_{\neg x}$) est ϕ dans laquelle toutes les clauses contenant x (resp. $\neg x$) sont satisfaites et retirées, et dans laquelle le littéral $\neg x$ (resp. x) est retiré des clauses restantes. La valeur MinSAT de l'instance initiale ϕ (i.e. le nombre minimal de clauses satisfaites de ϕ) est #soft(ϕ)-MinSatz(ϕ, 0), où #soft(ϕ) est le nombre de clauses molles de ϕ.

Prenons le temps maintenant de décrire comment est calculé UB. Supposons que l'on se trouve sur un nœud de l'espace de recherche et que, après avoir appliqué la propagation unitaire sur les clauses dures uniquement, on se trouve devant une instance formée des clauses dures $\{h_1, \ldots, h_k\}$, et des clauses molles $\{c_1, \ldots, c_m\}$ (non encore décidées). On construit alors un graphe non dirigé $G = (V, E)$ où V contient un nœud pour chaque clause dans $\{c_1, \ldots, c_m\}$ (que l'on notera $V = \{v_1, \ldots, v_m\}$). Un arc est ajouté entre v_i et v_j si la clause c_i contient un littéral l, et si la clause c_j contient $\neg l$.

De plus, un arc entre v_i et v_j est ajouté dès que l'on peut prouver, par propagation unitaire, la contradiction sur l'ensemble des littéraux $\{\neg l_1^i, \ldots, \neg l_p^i, \neg l_1^j, \ldots, \neg l_q^j, h_1, \ldots, h_k\}$, avec $c_i = \{l_1^i, \ldots, l_p^i\}$ (correspondant au nœud v_i) et $c_j = \{l_1^j, \ldots, l_q^j\}$ (associé au nœud v_j). Intuitivement, la représentation à l'aide du graphe G permet d'exprimer le fait que les clauses molles associées à deux arcs d'un même nœud ne peuvent pas être simultanément falsifiées. Dans le premier cas, il existe en effet au moins une clause molle satisfaite dans la mesure ou soit l, soit $\neg l$ est satisfait par toute assignation. Dans le second cas, si les clauses molles associées à deux arcs d'un nœud sont falsifiées, alors, par construction, cela impliquerait la falsification d'une clause dure.

Une fois que le graphe G est construit, une partition en cliques est construite à l'aide de l'algorithme décrit dans [Li and Quan 2010b, Li and Quan 2010a]. Par construction de G, il existe au plus une clause falsifiée dans chaque clique, et au moins toutes les clauses de chaque clique, sauf une, sont satisfaites par toute assignation complète. Le nombre de cliques d'une partition, noté s, est donc une surestimation du nombre de clauses molles qui peuvent être falsifiées par l'assignation partielle courante, par rapport à toute extension complète de celle-ci. En se basant sur ce point, on peut définir $UB = e + s$, où e est le nombre de clauses molles vides dans l'instance MinSAT courante.

On peut noter que le graphe aurait aussi pu être défini de telle manière qu'un arc entre v_i et v_j aurait été ajouté dès que l'ensemble de clauses $\{\neg l_1^i, \ldots, \neg l_p^i, \neg l_1^j, \ldots, \neg l_q^j, h_1, \ldots, h_k\}$ aurait pu être prouvé insatisfiable en utilisant un SAT solveur complet comme une boîte noire, plutôt qu'en se basant sur la propagation unitaire. Cela aurait certainement conduit à de meilleurs calculs de bornes mais aurait aussi, en contrepartie, eu un impact sur la rapidité du solveur. Nous nous sommes donc restreint dans ce travail aux contradictions détectables par propagation unitaire.

Exemple 1 *Supposons que, lors de la recherche, l'on soit sur un nœud contenant les clauses dures* $\neg x_1 \vee \neg x_2$, $\neg x_2 \vee \neg x_3$, $\neg x_3 \vee \neg x_4$, $\neg x_4 \vee \neg x_5$, $\neg x_1 \vee \neg x_5$, *et les clauses molles* $\neg x_1, \neg x_2, \neg x_3, \neg x_4, \neg x_5$. *Supposons, de plus, qu'aucune clause n'a encore été réduite à vide. On construit alors le graphe G ayant les nœuds v_1, v_2, v_3, v_4, v_5, chaque nœud v_i étant associé aux clauses molles $\neg x_i$ (pour $1 \leq i \leq 5$). Les arcs de G sont $\{(v_1, v_2), (v_2, v_3), (v_3, v_4), (v_4, v_5), (v_1, v_5)\}$, ce qui correspond au graphe représenté figure 1. Supposons maintenant que notre algorithme trouve la partition en cliques de G suivante : $\{\{v_1, v_2\}, \{v_3, v_4\}, \{v_5\}\}$. Alors, au plus 3 clauses molles sur les 5 clauses initiales peuvent être réduites à vide, ce qui nous donne $UB = 3$.*

Cependant, il faut bien noter que la partition ne peut donner une borne supérieure optimale si G n'est pas parfait, comme l'exemple suivant, adapté de celui donné dans [Li and Quan 2010b], le montre bien.

En effet, le graphe figure 1 peut être partitionné en trois cliques $\{(v_1, v_2), (v_3, v_4), (v_5)\}$, donnant dès

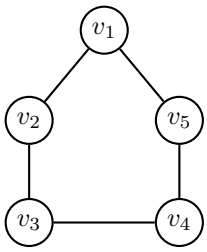


FIG. 1 – Un graphe imparfait simple ($\chi(G)$=3 et $\omega(G)$=2)

lors une borne supérieure de 3 pour le nombre de clauses falsifiées. Néanmoins, une analyse plus poussée montre que seulement deux clauses peuvent être falsifiées (au lieu de 3). Cette analyse peut par exemple se baser sur une adaptation de l'approche proposée dans [Li and Quan 2010b, Li and Quan 2010a] pour transformer la partition en cliques de G en une instance MaxSAT partielle. Dès lors, il devient possible d'utiliser un solveur MaxSAT pour améliorer la borne supérieure. Le raisonnement, en logique propositionnelle, de ce solveur serait équivalent au suivant : Supposons que toute clique contienne une clause falsifiée pour toute assignation complète. Alors, v_5 devrait être falsifié, mais v_1 et v_4 ne le peuvent, puisque v_1 et v_4 sont connectés à v_5. Donc, la seule possibilité pour la première et la seconde clique de contenir une clause falsifiée serait que v_2 et v_3 soient tous deux falsifiés, ce qui est impossible dans la mesure où v_2 et v_3 sont connectés. Donc, $\{(v_1, v_2), (v_3, v_4), (v_5)\}$ est un sous-ensemble de cliques qui ne permet pas à toutes ses cliques d'avoir une clause falsifiée en même temps.

Dans le cas général, on utilise les techniques liées à MaxSAT pour détecter des sous-ensembles de cliques qui ne peuvent avoir en même temps des clauses falsifiées, comme dans [Li and Quan 2010b, Li and Quan 2010a]. Chaque sous-ensemble permet d'améliorer la borne supérieure de un. Le lecteur est invité à se référer aux deux articles précédemment cités pour de plus amples détails sur ces techniques. La détection demande un temps en $O(m^2)$ pour la partition de graphe (pour m clauses molles), ce qui induit une complexité en temps de la fonction overestimation(ϕ) en $O(m^2)$ également. La représentation du graphe demande par ailleurs un espace en $O(m^2)$ et les cliques obtenues sont stockées dans un espace en $O(m)$.

Dans le cas pondéré, lorsque tous les poids sont des entiers positifs, on définit le poids d'un nœud v de G comme étant le poids de la clause molle associée c. Soit $\mathcal{P}$ l'ensemble des cliques de G, tel que deux cliques ont la possibilité de partager un nœud, et tel que chaque clique clq_i est associée à un poids w_{clq_i}. On appelle $\mathcal{P}$ la *Partition pondérée en cliques* de G si pour tout nœud v de G, son poids est égal à $\sum_{v \in clq_i} w_{clq_i}$. Par définition, chaque nœud de G appartient à au moins une clique de $\mathcal{P}$.

Exemple 2 *Dans la figure 1, supposons que les cinq clauses molles pondérées correspondant aux cinq noeuds soient {(c_1, 2), (c_2, 3), (c_3, 4), (c_4, 5), (c_5, 6)}. Alors $\mathcal{P}_1$={{(v_1, v_2), 2}, {(v_3, v_4), 4}, {(v_5), 6}, {(v_2), 1}, {(v_4), 1}} et $\mathcal{P}_2$={{(v_1, v_2), 2}, {(v_2, v_3), 1}, {(v_3, v_4), 3}, {(v_4, v_5), 2}, {(v_5), 4}} sont des partitions pondérées en cliques du graphe.*

Soit une instance MinSAT ϕ, une borne supérieure triviale pour la somme des poids des clauses molles falsifiées est $UB = \sum_{c_i \in \phi} w_i$, où w_i est le poids de la clause c_i. Chaque clique $\{(v_{i_1}, \ldots, v_{i_k}), w\}$ de poids w permet d'améliorer cette borne de (k-1)w, dans la mesure où cette clique implique au moins k-1 clauses satisfaites. En d'autres termes, chaque clique permet de ne pas compter, dans la borne supérieure, le poids w des k-1 clauses molles. Dans l'exemple 2, la partition pondérée en cliques $\mathcal{P}_1$ permet d'obtenir une borne supérieure de 14, alors que $\mathcal{P}_2$ donne une borne supérieure de 12.

La génération d'une partition pondérée en cliques d'un graphe pondéré G revient à partitionner les poids de tous les nœuds en un ensemble de cliques de G. C'est ce que permet de faire l'algorithme 2, utilisé dans MinSatz. Tout d'abord, cet algorithme partitionne G sans prendre en compte les poids associés. Ensuite, il cherche à associer à chaque clique $\{v_{i_1}, \ldots, v_{i_k}\}$ le poids $w = \min(w_{i_1}, \ldots, w_{i_k})$, où w_{i_j} est le poids de v_{i_j} dans G pour $1 \leq j \leq k$. Il modifie ensuite le poids de v_{i_j} dans G pour qu'il soit $w_{i_j} - w$. Une fois que toutes les cliques ont été associées à un poids, un nouveau graphe G' est construit d'après G, en ôtant tous les nœuds de poids nul. Ce mécanisme se poursuit ensuite pour G'. On peut noter que s'il y a t cliques dans la partition de G, G' a au moins t nœuds de moins que G.

Algorithm 2: partition(ϕ)

Construit un graphe pondéré G associé à ϕ ;
$\mathcal{P} \leftarrow \{\}$;
repeat
  Trouve une partition en cliques de G, et ajoute les cliques à $\mathcal{P}$;
  Construit G' d'après les G et les cliques obtenues ;
  $G \leftarrow G'$;
until G devienne vide
Retourner $\mathcal{P}$.

Dans l'exemple 2, $\mathcal{P}_1$ est obtenu en utilisant l'algorithme 2. On notera que $\mathcal{P}_2$ est de meilleur qualité que $\mathcal{P}_1$. Bien entendu, trouver de manière efficace une partition pondérée en cliques de meilleur qualité va permettre d'améliorer de manière significative la recherche restant à faire.

L'utilisation des technologies MaxSAT va aussi permettre d'améliorer encore la borne supérieure donnée par

la partition ci-dessus, en suivant la même idée que pour le cas non pondéré. Chaque fois que l'on détecte un sous-ensemble de cliques dans lequel toutes les cliques ne peuvent avoir une clause falsifiée en même temps, la borne supérieure peut être améliorée de w, où w est le poids minimum sur toutes les cliques du sous-ensemble.

4 Évaluation expérimentale

L'évaluation expérimentale tient une part importante dans notre travail. Nous avons ainsi conduit cinq expérimentations pour nous permettre d'évaluer les performances et l'utilité pratique de MinSatz. Les expérimentations, sauf mention contraire, ont été réalisées sur un Macpro 2.8GHz Intel Xeon ayant 4Go de mémoire sous Mac OS X 10.5.

4.1 Benchmarks

MaxClique. L'encodage en problème MaxSAT partiel du problème MaxClique d'un graphe $G = (V, E)$, tel qu'utilisé dans les évaluations MaxSAT est le suivant [Heras *et al.* 2008] : Une variable propositionnelle est associée à chaque nœud de V. La variable x_i est vraie ssi le nœud v_i appartient à la clique. Pour chaque paire v_i, v_j de nœuds non adjacents, correspond une clause dure $\neg x_i \vee \neg x_j$. Pour chaque nœud v_i, il y a une clause molle unitaire $(x_i, 1)$. Résoudre l'instance correspondante revient à chercher à maximiser le nombre de nœuds appartenant à la clique.

On peut obtenir très simplement un encodage MinSAT partiel de ce même problème MaxClique en utilisant les mêmes clauses dures et, pour chaque nœud v_i, en utilisant plutôt une clause molle unitaire $(\neg x_i, 1)$. Dès lors, résoudre l'instance obtenue revient à minimiser le nombre de nœuds n'appartenant pas à la clique.

Enchères combinatoires. Pour rappel, le problème des enchères combinatoires se définit comme suit. On a un ensemble de biens G, et un ensemble d'acheteurs qui peuvent enchérir sur des ensembles indivisibles de biens. Chaque enchère b_i est définie comme le sous-ensemble des biens voulus $G_i \subseteq G$ et le montant proposé pour l'achat. Le vendeur, qui veut bien entendu maximiser ses revenus, doit décider quelles enchères il désire satisfaire, dans la mesure où deux enchères ayant un bien commun ne peuvent être simultanément acceptées. L'encodage en problème MaxSAT partiel pondéré de ce problème, tel que proposé dans l'évaluation MaxSAT est le suivant [Heras *et al.* 2008] : Une variable propositionnelle est associée à chaque enchère. x_i est vraie ssi l'enchère b_i est acceptée. Pour chaque paire d'enchères (b_i, b_j) contenant au moins un bien commun, une clause dure $\neg x_i \vee \neg x_j$ est ajoutée (encodant ainsi le fait qu'on ne puisse pas accepter ces deux enchères simultanément). Pour chaque enchère b_i, est ajouté une clause unitaire molle (x_i, w_i) représentant le profit w_i potentiel associé à l'acceptation de l'enchère b_i. Ainsi, résoudre l'instance revient à maximiser les profits obtenus.

L'encodage sous forme de problème MinSAT partiel pondéré s'obtient en utilisant les mêmes clauses dures que précédemment, mais en considérant plutôt les pertes de profits associées au refus de chaque enchère. Ainsi, pour chaque enchère b_i est ajouté une clause unitaire molle $(\neg x_i, w_i)$ indiquant une perte de profit de w_i si b_i n'est pas acceptée. Dans ce cas, résoudre l'instance revient naturellement à minimiser les pertes.

Min-3SAT. Rappelons que le problème Min-3SAT (non pondéré) consiste à résoudre une instance MinSAT dont chaque clause contient exactement 3 littéraux. MinSAT peut aussi se réduire à MaxSAT partiel. Trois encodages (E1, E2, et E3) ont été présentés dans [Li *et al.* 2010b].

4.2 Solveurs

Nous avons comparé MinSatz aux solveurs suivants :

- akmaxsat, akmaxsat_ls [Kuegel 2010], IncMaxSatz [Lin *et al.* 2008], MaxSatz, Wmaxsatz+ [Li *et al.* 2007, Li *et al.* 2010a] : solveurs MaxSAT partiel pondéré, basés sur des techniques branch-and-bound, et soumis à l'évaluation MaxSAT 2010.
- MaxCLQ [Li and Quan 2010a] et MaxCliqueDyn (Noté MCQDyn) [Konc and Janezic 2007] : les deux meilleurs solveurs spécifiques au problème MaxClique, à notre connaissance.
- CASS [Fujishima *et al.* 1999] : solveur spécifique (état de l'art) pour le problème des enchères combinatoires.
- CPLEX : solveur commercial (version 8.0) utilisant des techniques de programmation linéaire.

4.3 Analyse des résultats expérimentaux

Toutes les tables reportent des temps d'exécutions moyens en secondes, calculés sur celles ayant terminé avant la limite de temps donnée. Ces temps sont donc suivis du nombre d'instances effectivement résolues.

Dans la première expérimentation, nous avons cherché à résoudre les problèmes Min-3SAT aléatoires de [Li *et al.* 2010b], en utilisant une limite de temps de 3 heures (identique à [Li *et al.* 2010b]). Nous comparons ici les performances de MinSatz avec l'encodage en MaxSat partiel proposé dans l'article cité. Les résultats sont résumés table 1. Notons que ces instances ont été générées uniformément, de 40 à 100 variables, et avec un nombre de clauses correspondant à des ratios clauses-sur-variables (C/V) prédéterminés (4, 4.25 et 5).

La table 1 montre clairement combien un solveur MinSAT dédié est plus performant que le meilleur solveur MaxSAT. On notera que MinSatz a résolu toutes les ins-

$instance$		$MinSatz$	$MaxSatz$		
#var	C/V		$E1$	$E2$	$E3$
40	4.00	0.01*(50)*	3.28*(50)*	507.8*(50)*	0.19*(50)*
50	4.00	0.03*(50)*	49.30*(50)*	- *(0)*	0.92*(50)*
60	4.00	0.12*(50)*	742.4*(50)*	- *(0)*	4.96*(50)*
70	4.00	0.41*(50)*	5735*(34)*	- *(0)*	23.21*(50)*
80	4.00	1.97*(50)*	- *(0)*	- *(0)*	100.7*(50)*
90	4.00	8.87*(50)*	- *(0)*	- *(0)*	381.5*(50)*
100	4.00	30.59*(50)*	- *(0)*	- *(0)*	1658*(33)*
40	4.25	0.01*(50)*	4.67*(50)*	992.5*(50)*	0.28*(50)*
50	4.25	0.04*(50)*	75.6*(50)*	- *(0)*	1.57*(50)*
60	4.25	0.14*(50)*	1153*(50)*	- *(0)*	8.31 *(50)*
70	4.25	0.69*(50)*	5989*(5)*	- *(0)*	42.77*(50)*
80	4.25	3.02*(50)*	- *(0)*	- *(0)*	186.3*(50)*
90	4.25	10.52*(50)*	- *(0)*	- *(0)*	760.4*(50)*
100	4.25	61.85*(50)*	- *(0)*	- *(0)*	2819*(26)*
40	5.00	0.02*(50)*	10.87*(50)*	5693*(48)*	0.80*(50)*
50	5.00	0.07*(50)*	226.8*(50)*	- *(0)*	5.24*(50)*
60	5.00	0.47*(50)*	3803*(48)*	- *(0)*	39.3*(50)*
70	5.00	1.59*(50)*	- *(0)*	- *(0)*	243.4*(50)*
80	5.00	14.68*(50)*	- *(0)*	- *(0)*	1512*(50)*
90	5.00	75.57*(50)*	- *(0)*	- *(0)*	5167*(39)*
100	5.00	380.72*(50)*	- *(0)*	- *(0)*	- *(0)*

TAB. 1 – Temps moyens (en secondes) de résolution pour les instances Min-3SAT résolues en 3 heures, suivi par le nombre d'instances effectivement résolues (sur 50)

tances dans le temps imparti, ce qui n'est pas le cas de MaxSatz.

Dans l'expérimentation suivante, nous avons cherché à résoudre les 96 problèmes aléatoires de MaxClique utilisés dans l'évaluation MaxSAT 2010[1], ainsi que les 66 [2] instances de DIMACS (dont 62 sont aussi dans l'évaluation sous le nom *structured*). Le temps maximal est aligné sur celui de l'évaluation (1800s). Nous comparons MinSatz avec les deux meilleurs solveurs MaxSAT sur ces instances : IncMaxSatz et akmaxsat, ainsi qu'avec les deux meilleurs solveurs spécifiques pour MaxClique (MaxCLQ et MCQDyn). Cette expérimentation a été conduite sur une machine Intel core 2 duo à 3.33 GHz de 4Go de mémoire sous Linux, dans la mesure où les solveurs MaxSAT étaient disponibles en binaires pour ce système seulement. Les résultats sont résumés table 2. Clairement, MinSatz obtient des performances bien supérieures aux deux solveurs MaxSAT et arrive à égaliser les performances des solveurs spécifiques. Dans la mesure où le problème MaxClique, en soit, est étudié depuis longtemps, nous pensons que cela montre qu'une réduction à MinSAT est une solution alternative pour toute une série de problèmes d'optimisations.

Dans la troisième expérimentation, nous avons chercher à résoudre les problèmes d'enchères combinatoires utilisées dans l'évaluation MaxSAT 2010. Ils ont été générés à partir de la suite de problèmes *Combinatorial Auction Test Suite* (CATS) [Leyton-Brown *et al.* 2000], basée sur un générateur aléatoire s'inspirant d'exemples réels. Nous avons utilisés les ensembles *Paths* (88 instances) et *Scheduling* (84 instances). Encore une fois, nous comparons MinSatz avec les résultats obtenus lors de l'évaluation MaxSAT 2010, mais avec les solveurs akmaxsat, akmaxsat_ls, et Wmaxsatz+, qui sont les meilleurs sur ces instances. Le temps maximal est de 1800s. Les résultats en temps de ces solveurs sont ceux de l'évaluation, obtenus sur une machine AMD Opteron de 2GHz, ayant 512Mo de mémoire, sous Linux. Les résultats sont donnés table 3. Même en prenant en compte la différence de puissance entre les machines utilisées pour les tests, il demeure clair que MinSatz est de plusieurs ordres de grandeur plus performants que ces solveurs MaxSAT. Dans la mesure où ces instances étaient finalement facile pour MinSatz, nous avons décidé de comparer notre approche avec les meilleurs solveurs dédiés du domaine.

instances	MinSatz	IncMaxsatz	akmaxsat	MaxCLQ	MCQDyn
random(96)	0.01*(96)*	1.6*(96)*	3*(96)*	0.01*(96)*	0.01*(96)*
DIMACS(66)	115*(53)*	88*(39)*	84*(40)*	69*(54)*	57*(52)*

TAB. 2 – Temps moyens (en secondes) pour MaxClique, dans un temps maximal de 1800s, suivi par le nombre d'instances résolues.

instances	MinSatz	akmaxsat_ls	akmaxsat	Wmaxsatz+
path (88)	0.01*(88)*	22*(88)*	23*(88)*	192*(71)*
scheduling (84)	0.01*(84)*	267*(75)*	230*(73)*	63*(83)*

TAB. 3 – Problèmes d'enchères combinatoires (temps en secondes)

Ceci nous mène donc à notre quatrième expérimentation, où nous comparons les performances de MinSatz avec CASS et CPLEX sur les instances d'enchères combinatoires disponibles à http ://people.cs.ubc.ca/˜ kevinlb/downloads.html. Ces instances sont générées par CATS 2.0 pour 9 distributions. Nous avons utilisé la suite "variable problem size" ayant de 40 à 400 biens et de 50 à 2000 enchères. Chaque distribution contient 800 instances, mais, par soucis d'économie, nous avons arbitrairement restreint notre étude aux 100 premières instances (numérotées 000001 à 000100). Les résultats sont reproduits table 4, en utilisant cette fois un temps limite de 3600s. Les temps de calcul de CASS et de CPLEX pour ces instances, obtenus sur une machine Xeon 2.4GHz, sont reproduits depuis cette même page internet. Pour tenter de comparer au mieux les différentes approches, nous avons divisé les temps de CASS et CPLEX pour prendre en compte la différence d'architecture machine. MinSatz est plus rapide que CASS sur ces instances, mais plus lent que CPLEX. On notera cependant que, sur la distribution L7, MinSatz résout un

[1] http ://www.maxsat.udl.cat/10/introduction/index.html
[2] available at http ://cs.hbg.psu.edu/txn131/clique.html

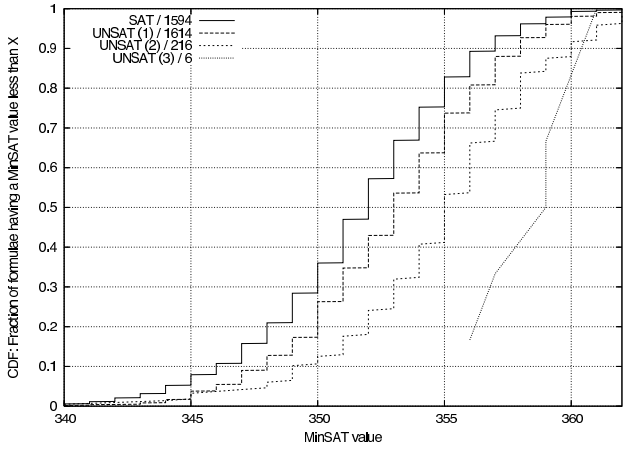


FIG. 2 – Courbes CDF pour 511 clauses et 120 variables (3430 essais). Les courbes sont respectivement basées sur 1594 instances (MaxSAT=511, i.e. toutes les 511 clauses de chaque instance peuvent être satisfaites), 1614 instances (MaxSAT=510, i.e. 510 seulement des 511 clauses peuvent être satisfaites), 216 instances (MaxSAT=509) et seulement 6 instances pour MaxSAT=508.

certain nombre d'instances que CPLEX n'arrive pas à résoudre dans le temps imparti.

Instances	MinSatz	CASS	CPLEX
Arbitrary(100)	108*(52)*	133*(39)*	34*(68)*
L2(100)	0.01*(100)*	0.01*(100)*	1*(100)*
L3(100)	164*(43)*	165*(32)*	161*(50)*
L4(100)	131*(52)*	84 *(46)*	32*(97)*
L6(100)	170*(44)*	90*(38)*	0.01*(100)*
L7(100)	56*(93)*	6*(88)*	154*(89)*
Matching(100)	98*(79)*	148*(31)*	0.01*(100)*
Region(100)	186*(50)*	37*(35)*	17*(96)*
Scheduling(100)	112*(82)*	28*(71)*	2*(100)*
Total(900)	98*(595)*	44*(480)*	49*(800)*

TAB. 4 – Problèmes difficiles d'enchères combinatoires (temps en secondes)

Il faut tout de même rappeler ici combien CPLEX est un logiciel hautement optimisé, de même que son encodage du problème des enchères combinatoires. De notre point de vue, il ne fait pas de doutes que MinSatz puisse être encore optimisé, à la fois pour les problèmes généraux MinSAT, mais aussi pour les problèmes plus spécifiques d'enchères combinatoires. Ainsi, la borne supérieure pourrait être optimisée à l'aide d'une meilleure partition pondérée du graphe ou grâce à des techniques comme la consistance par arcs virtuels [Cooper *et al.* 2010].

Dans la cinquième – et dernière – expérimentation, nous avons étudié l'existence d'éventuelles relations entre la valeur MaxSAT d'une instance aléatoire donnée et sa valeur MinSAT. Ce point est d'ailleurs la raison originelle ayant conduit au travail autour de MinSAT.

Nous avons généré un ensemble de formules 3SAT aléatoires ayant 120 variables au seuil (avec 511 clauses). La valeur MaxSAT de ces instances a permis de les partitionner, chaque partition contenant toutes les formules 3SAT d'une même valeur MaxSAT. La figure 2 montre clairement que pour les 4 partitions obtenues, la courbe CDF (Cumulative Distribution Function) est significativement différente, suggérant que la distribution des valeurs MinSAT est distincte entre partitions. On notera qu'un point (x, y) sur ces courbes signifie que la fraction des instances ayant une valeur MinSAT inférieure à x dans la partition correspondante est y. En d'autres termes, y peut grossièrement se voir comme la probabilité qu'une instance ayant une certaine valeur MaxSAT ait une valeur MinSAT inférieure ou égale à une certaine valeur. De manière à confirmer cette observation, nous avons procédé à des tests de type *two-sample Kolmogorov-Smirnov* entre tous les couples de courbes. L'hypothèse nulle a été rejetée avec une quasi certitude.

Cette dernière expérimentation permet de conclure sur une observation relativement surprenante. En effet, il semble que les instances aléatoires ayant la plus grande valeur MaxSAT sont aussi celles qui ont tendance, de manière significative, à avoir la valeur MinSAT la plus petite. Nous avons aussi pu confirmer ce phénomène avec des ratio r=C/V plus élevés (jusque $r = 8$) et aussi pour un grand nombre d'instances de 80 et 100 variables à différents ratios.

5 Conclusion

Nous avons proposé, dans cet article, et pour la première fois, de voir le problème MinSAT d'un point de vue applicatif et pragmatique. Nous avons proposé le premier solveur MinSAT pondéré partiel, appelé MinSatz. Bien que MinSAT et MaxSAT soient tous deux des extensions naturelles de SAT, l'utilité de MinSAT, en pratique, n'était pas claire avant nos travaux. De plus, bien que les encodages MaxSAT et MinSAT utilisés soient très similaires, il apparaît clairement combien l'utilisation du formalisme MinSAT est extrêmement compétitive par rapport à MaxSAT. Nous pensons que cela est dû au fait que notre solveur permet d'intégrer des mécanismes de calculs de bornes mêlant raisonnement en logique propositionnelle et partitions de graphes, qui sont fondamentalement différents de ceux utilisés par les solveurs MaxSAT, et qui semblent être particulièrement performants pour exploiter les structures de graphe. Nos expérimentations montrent aussi que notre approche est une alternative de choix pour certains problèmes d'optimisations, dans la mesure où les performances affichées dépassent les solveurs dédiés aux problèmes testés (MaxClique et Enchères combinatoires). L'efficacité obtenue nous a aussi permis d'étudier expérimentalement l'existence d'une corrélation possible entre

les valeurs MinSAT et MaxSAT des instances 3SAT aléatoires uniformes.

Dans la suite de ce travail, nous pensons améliorer les techniques de calculs de bornes en utilisant une meilleure partition des graphes considérés, et pousser plus loin l'étude des performances de MinSatz sur les instances d'enchères combinatoires.

Références

[Avidor and Zwick 2005] Adi Avidor and Uri Zwick. Approximating min 2-sat and min 3-sat. *Theory of Computing Systems*, 38(3) :329–345, 2005.

[Cooper *et al.* 2010] M. Cooper, S. de Givry, M. Sanchez, T. Schiex, M. Zytnicki, and T. Werner. Soft arc consistency revisited *Artificial Intelligence*, (7-8) :449-478, 2010

[Fujishima *et al.* 1999] Yuzo Fujishima, Kevin Leyton-Brown, and Yoav Shoham. Taming the computational complexity of combinatorial auctions : Optimal and approximate approaches. In *IJCAI*, pages 548–553, 1999.

[Heras *et al.* 2008] Federico Heras, Javier Larrosa, Simon de Givry, and Thomas Schiex. 2006 and 2007 Max-SAT evaluations : Contributed instances. *JSAT*, 4(2-4) :239–250, 2008.

[Kohli *et al.* 1994] Rajeev Kohli, Ramesh Krishnamurti, and Prakash Mirchandani. The minimum satisfiability problem. *SIAM J. Discrete Mathematics*, 7(2) :275–283, 1994.

[Konc and Janezic 2007] J. Konc and D. Janezic. An improved branch and bound algorithm for the maximum clique problem. *Communications in Mathematical and in Computer Chemistry*, 58 :569–590, 2007.

[Kuegel 2010] Adrian Kuegel. Improved exact solver for the Weighted MAX-SAT problem. In *Workshop Pragmatics of SAT*, 2010.

[Leyton-Brown *et al.* 2000] Kevin Leyton-Brown, Mark Pearson, and Yoav Shoham. Towards a universal test suite for combinatorial auction algorithms. In *ACM Conference on Electronic Commerce*, pages 66–76, 2000.

[Li and Anbulagan 1997] Chu Min Li and Anbulagan, Heuristics Based on Unit Propagation for Satisfiability Problems. In Proceedings of IJCAI, pages 366–371,

[Li and Manyà 2009] Chu Min Li and F. Manyà. MaxSAT, hard and soft constraints. In Armin Biere, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, pages 613–631. IOS Press, 2009.

[Li and Quan 2010a] Chu Min Li and Zhe Quan. Combining graph structure exploitation and propositional reasoning for the maximum clique problem. In *ICTAI*, pages 344–351, 2010.

[Li and Quan 2010b] Chu Min Li and Zhe Quan. An efficient branch-and-bound algorithm based on maxsat for the maximum clique problem. In *AAAI*, pages 128–133, 2010.

[Li *et al.* 2007] Chu Min Li, Felip Manyà, and Jordi Planes. New inference rules for Max-SAT. *Journal of Artificial Intelligence Research*, 30 :321–359, 2007.

[Li *et al.* 2010a] Chu Min Li, Felip Manyà, Nouredine Ould Mohamedou, and Jordi Planes. Resolution-based lower bounds in MaxSAT. *Constraints*, 15(4) :456–484, 2010.

[Li *et al.* 2010b] Chu Min Li, Felip Manyà, Zhe Quan, and Zhu Zhu. Exact MinSAT solving. In *SAT*, pages 363–368, 2010.

[Lin *et al.* 2008] Han Lin, Kaile Su and Chu Min Li. Within-Problem Learning for Efficient Lower Bound Computation in Max-SAT Solving. In *AAAI*, pages 351–356, 2008.

[Marathe and Ravi 1996] M. V. Marathe and S. S. Ravi. On approximation algorithms for the minimum satisfiability problem. *Information Processing Letters*, 58 :23–29, 1996.

Intensification/Diversification VNS Application au Problème de Coloration de Graphe

Samir Loudni
GREYC (CNRS - UMR 6072) – Université de Caen
Boulevard du Maréchal Juin
14000 Caen
Samir.Loudni@unicaen.fr

Abstract

Étant donné un graphe non-orienté $G = (V, E)$, le problème de coloration consiste à déterminer le nombre minimal de couleurs nécessaires pour colorier les sommets de G de telle sorte que deux sommets adjacents n'aient pas la même couleur. Dans cet article nous présentons une extension de l'algorithme de base VNS, appelé `ID-VNS`, qui intègre des mécanismes simples pour réaliser un bon compromis entre intensification et diversification. Ensuite, nous montrons comment `ID-VNS` peut être appliqué avec succès au problème de coloration en définissant des voisinages génériques qui bénéficient à la fois des avantages des conflits et de la topologie du graphe de contraintes. Des expérimentations réalisées sur les instances difficiles de DIMACS montrent que notre méthode est très compétitive avec les meilleurs algorithmes connus de coloration.

1 Introduction

Étant donné un graphe non-orienté $G = (V, E)$, avec un ensemble V de sommets et un ensemble E d'arêtes, le *problème de coloration* des sommets d'un graphe (PCSG) consiste à déterminer le nombre minimum de couleurs (*nombre chromatique* $\chi(G)$) nécessaires pour colorier les sommets de G tel qu'il n'existe pas deux sommets adjacents de la même couleur. Déterminer une k-coloration est un problème NP-complet pour un nombre de couleurs $k \geq 3$ [13]. Ce problème a été largement étudié dans la littérature, non seulement pour ses applications directes dans des domaines variés, tels que l'allocation de registre [6], l'affectation de fréquences [12], les emplois du temps [4, 7] et l'ordonnancement [11, 19], mais aussi pour ses aspects théoriques et pour sa difficulté. En effet, bien que de nombreux algorithmes exactes ont été proposés pour ce problème [3, 5, 16, 24, 25], ces derniers ne peuvent résoudre que de petites instances (jusqu'à 100 sommets). Par conséquent, les méthodes heuristiques sont nécessaires pour traiter les grandes instances. Les approches heuristiques les plus performantes sont les méthodes de recherche locale (e.g., [1, 2, 17, 18, 27]) et les méthodes *hybrides à base de populations* (e.g. [8, 9, 22]).

La recherche à voisinages variables (VNS) [26] exploite l'idée de changer systématiquement de structure de voisinage, tant dans la descente aux minima locaux, mais aussi pour s'échapper des vallées qui les contiennent. Elle consiste en l'application successive d'une étape de *perturbation*, suivie d'une recherche locale. L'étape de perturbation joue un rôle primordial, car elle permet à VNS d'explorer différentes régions de l'espace de recherche, afin de s'échapper du bassin d'attraction des optima locaux (*effort de diversification*), alors que l'objectif de la recherche locale est de se concentrer plus intensivement au sein de chaque région pour converger vers un minimum local (*effort d'intensification*). Cependant, comme récemment mentionné dans [20], la plupart des algorithmes de RL considèrent diversification et intensification comme deux objectifs antagonistes : à mesure qu'on effectue plus d'intensification, on peut perdre en diversification, et vice-versa. Une plus grande coordination entre ces deux objectifs est donc nécessaire, en équilibrant entre (i) *degré d'exploration*, correspondant à la quantité d'efforts consacrées pour conduire l'exploration vers des régions distinctes de l'espace de recherche, et (ii) *degré d'exploitation*, définie par la quantité d'efforts déployée en direction de la recherche locale pour explorer une région prometteuse.

La liste de mouvements candidats (*Candidate List Strategy*) de type `Aspiration Plus` [15] est un mécanisme simple qui a été proposé pour la recherche tabou. Elle permet de restreindre le nombre de voisins examinés afin d'équilibrer entre l'effort de recherche dans le voisinage et la qualité du meilleur voisin trouvé. `IDWalk` [28] est une autre métaheuristique qui utilise une CLS pour gérer l'exploration d'un voisinage.

Dans cet article, nous présentons une extension de l'algorithme de base VNS, appelé `ID-VNS`, qui intègre des mécanismes simples pour réaliser un contrôle efficace de balance diversification/intensification au sein de VNS. Pour atteindre cet objectif, nous définissons tout d'abord une *mesure de diversité*, appelée *degré de perturbation*, pour contrôler la quantité de changements nécessaires pour transformer une solution en une autre. Deuxièmement, comme dans [28], nous utilisons une CLS pour contrôler l'effort d'exploration d'un voisinage par la recherche locale. Ensuite, nous montrons comment `ID-VNS` peut être appliquée au problème de coloration de graphe, en définissant des voisinages génériques qui bénéficient à la fois des avantages des conflits et de la topologie du graphe de contraintes.

Nous avons effectué des expérimentations sur plusieurs familles de graphes du deuxième "challenge" de DIMACS [29], et nous avons comparé nos résultats à ceux obtenus par six méthodes de recherche locale, ainsi que deux algorithmes hybrides à base de populations HCA [9] et MMT [22]. Ces résultats montrent que notre méthode domine nettement les méthodes de recherche locale et est très compétitive avec les approches hybrides.

La section 2 donne un aperçu synthétique du problème de coloration et présente les meilleurs algorithmes de coloration. Dans la section 3, nous décrivons notre méthode de résolution ID-VNS. Ensuite, nous détaillons (section 4) nos structures de voisinages. La section 5 est consacrée aux expérimentations. Enfin, nous concluons et donnons quelques perspectives.

2 Problème de coloration de graphe

2.1 Définitions et notations

Soit $G = (V, E)$ un graphe connexe avec un ensemble V de sommets et un ensemble E d'arêtes. Étant donné un entier positif k, une k-coloration de G est une fonction $c : V \rightarrow \{1, \ldots k\}$, qui attribue une valeur $c(u)$, appelée *couleur* de u, à chaque sommet $u \in V$. Si deux sommets adjacents u et v ont la même couleur, on dit qu'ils sont en *conflit* et que l'arête $(u, v) \in E$ est une arête *conflictuelle*.

Une k-coloration sans conflit est dite *légale*, sinon elle est *illégale*. Soit $s = [c(1), \ldots, c(|V|)]$ une k-coloration légale, s peut être représentée par une *partition* de V en k sous-ensembles disjoints V^1, ..., V^k. Nous notons par V^r la classe de couleur r induite par s (i.e. l'ensemble de sommets ayant la couleur r dans s).

Soit f la fonction objectif qui associe à chaque coloration s le nombre d'arêtes conflictuelles induites par s. Le problème de coloration des sommets d'un graphe (PCSG) consiste à déterminer le plus petit entier k, appelé *nombre chromatique* $\chi(G)$, tel qu'il existe une k-coloration légale de G.

2.2 Approches métaheuristiques

Dans cette section, nous décrivons brièvement les meilleurs algorithmes de coloration (voir [23] pour plus de détails).

Un des premiers algorithmes de recherche locale proposé pour ce problème est TABUCOL [17]. Depuis, il a été amélioré et incorporé avec succès dans plusieurs algorithmes complexes de coloration [9, 18].

Morgenstern [27] a proposé, dans le cadre d'un Recuit Simulé, un algorithme distribué complexe, noté MOR, utilisant un codage à base de colorations légales *partielles* (i.e., certains sommets peuvent ne pas être colorés). Chaque sommet non coloré induit une pénalité et l'objectif est de minimiser la somme des degrés des sommets non colorés.

Galinier et Hao [9] ont proposé une approche hybride évolutionniste, notée HCA, incorporant TABUCOL comme recherche locale, et avec un croisement glouton très efficace à base de partitions.

Dans [2], deux algorithmes Tabou (DYN-P.COL et FOO-P.COL), utilisant un codage sous forme de colorations légales partielles ont été proposés. DYN-P.COL (resp. FOO-P.COL) utilise un réglage dynamique (resp. réactif) de la durée Tabou.

Algorithm 1: Pseudo-code de ID-VNS

```
fonction ID-VNS(maxIter, maxneigh, maxMoves, nextneigh) ;
début
    Soit N_i, (i = 1,i_max), un ensemble prédéfini de structures de
    voisinages ;
1   s ← genRandomSol();
2   iter ← 1, i ← 1, b ← b^i_max;
3   tant que (iter ≤ maxIter) faire
4       iter ← iter + 1;
5       s' ← Perturbation(s, i, b) ;
6       s'' ← LS(s', maxneigh, maxMoves, nextneigh) ;
7       si f(s'') < f(s) alors
8           s ← s'', i ← 1;
9           b ← b^i_max, iter ← 1;
        sinon
10          si f(s'') = f(s) alors s ← s'' ;
11          si (iter mod α = 0) alors i ← i + 1 ;
12          b ← getB(iter, i);
13  retourner s;
```

Enfin, Hertz et al. [18] ont proposé une extension de l'algorithme VNS, appelé *Variable Search Space* (VSS-Col), dans lequel le processus de recherche commute itérativement entre trois espaces de recherche, chacun ayant son propre codage, sa propre fonction objective et son voisinage.

3 Intensification/Diversification VNS

3.1 Schéma de base de ID-VNS

ID-VNS étend l'algorithme de base VNS en incluant des mécanismes permettant de réaliser un contrôle efficace de balance diversification/intensification au sein de VNS. Pour parvenir à ce compromis, deux objectifs doivent être équilibrés : (i) *degré d'exploration* et (ii) *degré d'exploitation*. Pour le premier objectif, nous définissons une mesure de diversité, appelée *degré de perturbation* (b), pour contrôler la quantité de changements effectuées sur la solution courante, tout en conduisant la recherche locale vers les régions de l'espace de recherche non encore visités (cf. sec. 3.2). Pour le second objectif, nous utilisons une *liste de mouvements candidats* [15, 28] pour équilibrer l'intensification de la recherche dans le voisinage et la possibilité de s'échapper rapidement des optima locaux (cf. sec. 3.3).

Le pseudo-code de ID-VNS est décrit par l'algorithme 1. Nous désignons par i l'indice du voisinage courant N_i, et par b^i_{min} (resp. b^i_{max}) le nombre minimum (resp. maximum) de perturbations effectuées dans chaque N_i. L'algorithme part d'une solution initiale s générée aléatoirement. Le voisinage est initialisé à N_1 en ligne 2. Chaque itération de la boucle des lignes 3 à 12 consiste en l'application d'une perturbation de degré b à s dans le voisinage N_i (ligne 5), suivie d'une recherche locale appliquée à s' (ligne 6). Si la solution voisine obtenue s'' est de meilleure qualité que la solution courante s, alors s'' devient la solution courante et le voisinage est réinitialisé à N_1 (lignes 8). La perturbation dans N_i prend fin lorsque (iter/α) itérations sont effectuées sans améliorer s (ligne 11). L'algorithme considère alors une autre perturbation dans le voisinage suivant N_{i+1}. La valeur de α a été fixée à $(2 \times |V|/i_{max})$. l'algorithme s'arrête dès que maxIter itérations consécutives sont effectuées sans améliorer s.

3.2 Étape de perturbation

Chaque perturbation consiste à choisir une variable en conflit dans s et une valeur pour cette variable minimisant le nombre de conflits. Le choix de ces variables dépend de chaque N_i (cf. sec. 4). Une solution voisine s' est générée dans N_i, en réaffectant b variables en conflit.

Le paramètre b permet de contrôler le nombre d'éléments de la solution courante s qui sont modifiés. Il représente une mesure simple de *diversification*. En effet, la solution s' constitue un point de départ à partir duquel la recherche locale va explorer des régions de l'espace des solutions non encore visitées. Toutefois, le nombre de modifications que la perturbation devrait effectuer reste un point crucial. Il doit être suffisamment grand pour éviter à la recherche locale de retomber dans un optimum local déjà visité, tout en empêchant celle-ci à dégénérer en un algorithme multi-restart.

Nous proposons d'exploiter l'information sur l'effort d'amélioration de la solution courante pour ajuster dynamiquement la valeur du paramètre b. Dans notre approche, b diminue proportionnellement en fonction du nombre d'itérations consécutives effectuées sans amélioration de la solution courante (i.e., `iter`). Initialement, $b = b^i_{max}$. Durant la recherche, à chaque fois que s est améliorée b est réinitialisé à sa valeur de départ (ligne 9) ; sinon il est décrémenté à chaque fois que `iter` augmente, jusqu'à atteindre une valeur minimale b^i_{min} (ligne 12). L'objectif étant d'effectuer de fortes perturbations à chaque fois que s est améliorée, et de favoriser progressivement de petites perturbations lorsque s n'a pas été améliorée pendant une longue période. Après quelques expérimentations préliminaires, nous avons initialisé b^i_{min} à 20 pour $(i = 1, \ldots, 5)$ et b^i_{max} à 100 et 50, respectivement pour $i = 1$ et $i = 2 \ldots, 5$.

Algorithm 2: Pseudo-code de LS

```
   fonction LS (s, maxneigh, maxMoves, nextneigh) ;
   début
1    stopIter ← maxMoves;
2    s_meilleur ← ∅,  f(s_meilleur) ← ⊤;
3    iter ← 1,  s* ← s;
4    tant que (iter ≤ stopIter) faire
5      iter ← iter + 1;
6      nbtries ← 1;
7      premierTrouvé ← faux, trouvé ← faux;
8      tant que (nbtries ≤ maxneigh) et non(trouvé) faire
9        v ← randomConflictVertex(s);
10       s' ← getNeighbor(s, v);
11       si non(premierTrouvé) alors
12         premierTrouvé ← vrai;
13         s_premier ← s';
14       si (v, s'[v]) ∉ tabulist) ou (f(s') < f(s*) alors
15         s ← s';
16         trouvé ← vrai;
         sinon
17         si non(trouvé) et (nextneigh = best) et
           (f(s') < f(s_meilleur)) alors
18           s_meilleur ← s' ;
19       nbtries ← nbtries + 1;
20     si non(trouvé) et (nextneigh = first) alors
       s ← s_premier ;
21     si non(trouvé) et (nextneigh = best) alors
       s ← s_meilleur ;
22     insérer (v, c(v)) dans tabulist et rendre (v, c(v)) tabou
       pour T itérations;
23     si f(s) < f(s*) alors
24       stopIter ← iter + maxMoves;
25       s* ← s;
26   retourner s*
```

3.3 Étape de recherche locale

Comme recherche locale, nous avons utilisé un algorithme *tabou*. Son pseudo-code est décrit par l'algorithme 2. Étant donné une solution courante s, à chaque itération (boucle des lignes 8 à 19), l'algorithme génère une solution voisine s', en modifiant une variable en conflit v dans s et en calculant une nouvelle affectation x_v pour cette variable qui minimise les conflits (lignes 9 et 10). L'exploration du voisinage de s s'arrête dès qu'un voisin est accepté ou que `maxneigh` voisins ont été examinés. Pour évaluer les mouvements, nous utilisons des structures de données incrémentales [8, 10]. L'idée est de maintenir pour chaque paire (v, x_v) un *compteur de conflits* qui enregistre le nombre de variables dans s qui sont en conflit avec l'affectation $(v = x_v)$. À Chaque fois qu'un mouvement est effectué, uniquement les compteurs des variables affectées par ce mouvement sont mis à jour. Une fois le mouvement effectué, la variable v est interdite de revenir à sa précédente valeur pour les prochaines T itérations (ligne 22). Après quelques expériences préliminaires, la *durée tabou* T a été fixée à 20. L'algorithme s'arrête dès qu'un nombre fixé d'itérations (`stopIter`) a été effectué. Chaque fois qu'une meilleure solution est trouvée, `stopIter` est mis à jour en ligne 24 (effort d'intensification).

Pour contrôler finement le compromis *intensification/diversification* dans l'exploration des voisinages, il est nécessaire de contrôler (i) le nombre maximum de voisins à explorer pour choisir un mouvement et (ii) le choix du prochain voisin. En effet, le nombre de voisins à considérer devrait être assez grand pour se focaliser plus intensivement sur des régions jugées prometteuses (i.e., effort d'intensification). Toutefois, il devrait être suffisamment petit pour permettre à la recherche de s'échapper des optima locaux plus rapidement (i.e., effort de diversification). Dans ce cas, la manière de choisir le prochain voisin devrait guider la recherche vers des régions inexplorées de l'espace des solutions. Comme pour `IDWalk` [28], nous proposons d'intégrer à notre recherche tabou une *liste de mouvements candidats* pour contrôler le parcours d'un voisinage. Deux paramètres sont définis : `maxneigh` qui indique le nombre maximum de voisins à examiner avant d'effectuer un mouvement, et `nextneigh` qui précise quelle prochaine solution choisir si aucun voisin parmi les `maxneigh` visités n'a été accepté. Deux valeurs sont possibles pour ce paramètre : `first` : on choisit le premier voisin (`s_premier`) parmi les `maxneigh` candidats non acceptés, `best` : on choisit le voisin (`s_meilleur`) qui détériore le moins la fonction d'évaluation.

La boucle des lignes 8 à 19 s'arrête dès qu'une première solution s' qui améliore la meilleure solution connue s^* est obtenue ou aucune amélioration n'ait été trouvée après `maxneigh` itérations. Si la solution voisine obtenue s' n'est pas tabou ou qu'elle satisfait un *critère d'aspiration*[1], s' de-

1. Il s'agit d'éliminer le caractère tabou d'un mouvement lorsque celui-ci conduit à une solution meilleure que celle obtenue jusqu'ici.

vient la nouvelle solution courante (lignes 14 à 16) ; sinon, `s_meilleur` est mis à jour (ligne 18). Selon la valeur de `nextneigh`, la prochaine solution est sélectionnée parmi les candidats rejetés (lignes 20 et 21).

4 Structures de voisinages

Dans cette section, nous détaillons nos structures de voisinages génériques pour le problème de coloration de graphe. Ces voisinages tiennent compte de la topologie du graphe de contraintes et des conflits. On notera par $neighbors(u)$ l'ensemble des sommets adjacents à u et par $\mathcal{X}(s)$ l'ensemble des sommets en conflit dans la solution courante s.

4.1 ConflictVar

ConflictVar (`CV`) sélectionne aléatoirement un sommet v en conflit dans $\mathcal{X}(s)$, puis choisit une nouvelle classe de couleur V^i pour ce sommet qui *minimise* les conflits. Soit s_1 la solution obtenue. Si $f(s_1) > f(s)$, `CV` sélectionne aléatoirement un nouveau sommet parmi ceux en conflit dans $\mathcal{X}(s_1) \setminus \mathcal{X}(s)$ et lui assigne une nouvelle meilleure classe de couleur. Ce processus est répété jusqu'à ce qu'un mouvement *non-détériorant* (i.e., n'augmentant pas le nombre de conflits) est trouvé. Dans notre implémentation, nous évitons de changer la couleur d'un sommet plus d'une fois. Cette séquence de mouvements est appliquée avec b différents sommets en conflit. Un tel voisinage qui repose principalement sur un choix aléatoire, permet de diversifier la recherche.

4.2 Extensions de ConflictVar

`CV` choisit aléatoirement les sommets en conflit indépendamment de la topologie du graphe de contraintes. Les sommets sélectionnés peuvent être complètement isolées dans le graphe et peuvent avoir des degrés élevés. Dans un tel cas, il est peu probable de modifier la couleur d'un sommet sans créer de nombreux conflits, et donc de trouver une solution de meilleure qualité. Ainsi, nous proposons d'exploiter la topologie du graphe de contraintes pour favoriser la sélection de sommets qui sont le plus liés [21].

1. *ConflictVar-Connected* (`CV_C`) choisit aléatoirement un sommet origine v_{init} dans s et lui assigne une nouvelle meilleure couleur V^i possible. Soit s_1 la solution obtenue. Si $f(s_1) > f(s)$, `CV_C` sélectionne aléatoirement un nouveau sommet u parmi ceux en conflit dans $(\mathcal{X}(s_1) \setminus \mathcal{X}(s)) \cap V^i$ et lui assigne une nouvelle meilleure couleur V^j. Soit s_2 la nouvelle solution obtenue. Si $f(s_2) > f(s_1)$, `CV_C` sélectionne aléatoirement un nouveau sommet z parmi ceux en conflit dans $(\mathcal{X}(s_2) \setminus \mathcal{X}(s_1)) \cap V^j$ et lui assigne une nouvelle meilleure couleur V^r. Cette séquence de mouvements est répétée jusqu'à ce qu'un mouvement *non-détériorant* est trouvé, formant ainsi un *sous-graphe connexe* dans G. Nous évitons également de changer la couleur d'un sommet plus d'une fois. Ce processus est est réitéré avec b différents v_{init}. Le prochain v_{init} est choisi parmi ceux en conflit dans V^i.
2. *ConflictVar-Star* (`CV_S`) choisit aléatoirement un premier sommet "centre" v_c dans s et lui assigne une nouvelle meilleure couleur V^i possible. Ensuite, pour chaque sommet de $neighbors(v_c) \cap V^i$, une nouvelle couleur qui minimise les conflits est alors trouvée. Ce processus est est réitéré avec b différents v_c. Un nouveau centre est déterminé parmi les sommets en conflit dans $neighbors(v_c) \cap V^i$.
3. *ConflictVar-ConnectedStar* (`CV_CS`) choisit aléatoirement un premier sommet "centre-connexe" v_{cc} dans s et lui assigne une nouvelle meilleure couleur V^i possible. Ensuite, `CV_CS` considère chaque sommet en conflit dans $neighbors(v_{cc}) \cap V^i$ comme un centre à partir duquel `CV_S` est appliquée. Ce processus est est réitéré avec b différents v_{cc}. Un nouveau "centre-connexe" est déterminé parmi les sommets en conflit voisins des centres.
4. *Conflict ColorClass* (`C_CC`) sélectionne la classe de couleur V^i ayant le plus grand nombre de sommets en conflit (les égalités sont cassées aléatoirement), puis choisit, pour chacun de ces sommets, une autre meilleure couleur possible. Soit s_1 la nouvelle solution obtenue. Ensuite, b sommets nouvellement en conflit (i.e., ceux en conflit dans $\mathcal{X}(s_1) \setminus \mathcal{X}(s)$) sont alors sélectionnés, puis réaffectés à leur meilleure couleur possible. Une *liste tabou* est utilisée pour interdire la sélection de la classe V^i pour les prochaines l itérations de `ID-VNS`, avec $l = 0.6 \times |\mathcal{X}(s)| + rand(0, 9)$ [9], où $rand(0, 9)$ est une fonction retournant un nombre aléatoire dans $\{0, \ldots, 9\}$.

5 Résultats expérimentaux

Dans cette section, nous présentons des résultats détaillés sur différents types de graphe du challenge de DIMACS. Pour certains graphes difficiles, nous considérons un ensemble d'instances (G, k), avec différentes valeurs de k. En effet, la difficulté de colorier un graphe est strictement corrélée avec la valeur de k. Ainsi, le niveau de difficulté pour trouver une k-coloration légale peut varier de trivial à très difficile. Pour chaque instance, **nous nous comparons avec les meilleurs algorithmes de coloration**. Comme les expérimentations ont été menées sur diverses machines, nous rapportons (lorsque cela est possible), les temps CPU normalisés [2].

5.1 Instances DIMACS

Nous avons expérimenté notre algorithme sur les graphes difficiles suivants [29] :

- 12 graphes aléatoires classiques **DSJC**$n.y$ ayant n sommets et de densité égale à $0.y$. Nous avons considéré les graphes avec $n \in \{125, 250, 500, 1000\}$ et $y \in \{1, 5, 9\}$.
- 2 graphes géométriques aléatoires **DSJR**$n.r$ construits à partir d'un ensemble de n points aléatoires (dans le carré unité) en joignant chaque couple de points distants de moins d'un seuil de longueur (indiqué par r). Nous avons considéré les deux graphes avec $n = 500$ et $r \in \{1, 5\}$.
- 6 graphes **flat**n_x_0, qui sont des graphes quasi-aléatoires avec un nombre chromatique connu x. Les graphes flat sont générés en partitionnant l'ensemble des sommets en x classes et en ajoutant des arêtes uniquement entre les sommets de classes différentes. Nous avons

2. Pour une machine κ fois plus lente que la nôtre, les temps CPU rapportés seront divisés par κ.

Instance	k	N_i	MaxN.	Succ.	Time	Avg
DSJC250.5		N_1	200	18/20	113.5	0.1
n=250		N_2	175	18/20	148.1	0.1
m=15668	28	N_3	100	15/20	83.2	0.25
k^*=28		N_4	150	20/20	118.9	0
		N_5	**150**	**20/20**	**76.4**	**0**
DSJC250.9		**N_1**	50	20/20	8.5	0
n=250		N_2	100	20/20	9.8	0
m=27897	72	N_3	150	20/20	9.9	0
k^*=72		**N_4**	**50**	**20/20**	**8.4**	**0**
		N_5	50	20/20	10.86	0
DSJC500.1		N_1	175	17/20	142.4	0.15
n=500		N_2	175	17/20	448.9	0.2
m=24916	12	N_3	175	15/20	325.7	0.4
k^*=12		N_4	100	13/20	303.4	0.5
		N_5	**200**	**20/20**	**121.9**	**0**
	49	**N_1**	**100**	**16/20**	**965.4**	**0.3**
		N_2	175	14/20	889	1.15
		N_3	100	11/20	1140.2	0.75
DSJC500.5		N_4	200	16/20	1778.6	1.05
n=500		N_5	200	12/20	1894.3	0.95
m=125248		N_1	200	1/10	2077.6	1.5
k^*=48	48	**N_2**	**100**	**2/10**	**4762.6**	**1.2**
		N_3	100	2/10	6658.2	1.2
		N_4	100	2/10	6445.6	1.6
		N_5	50	1/10	2820.6	2.1
		N_1	**50**	**20/20**	**153.10**	**0**
	127*	N_2	100	20/20	173.3	0
		N_3	100	19/20	203.3	0.05
DSJC500.9		N_4	100	20/20	208.5	0
n=500		N_5	50	19/20	194.2	0.05
m=224874		N_1	**150**	**18/20**	**2451**	**0.15**
k^*=126	126	N_2	50	11/20	3513	0.45
		N_3	150	10/20	6072.4	0.5
		N_4	175	14/20	5348.8	0.3
		N_5	175	12/20	6093.4	0.4
DSJR500.1c		N_1	150	0/10	-	2
n=500		N_2	200	0/10	-	2
m=121275	85	N_3	50	0/10	-	2
k^*=85		N_4	50	1/10	152.2	1.8
		N_5	**50**	**10/10**	**1713.5**	**0**
		N_1	150	0/10	-	2.7
		N_2	100	0/10	-	3.1
	125	N_3	50	6/10	4407.9	0.4
DSJR500.5		N_4	100	0/10	-	1.4
n=500		**N_5**	**100**	**9/10**	**2702.6**	**0.15**
m=58862		N_1	175	0/10	-	3.6
k^*=122		N_2	100	0/10	-	3.9
	124	N_3	175	0/10	-	1.7
		N_4	175	0/10	-	1.8
		N_5	**175**	**1/10**	**297.1**	**1.4**
DSJC1000.1		N_1	50	20/20	3.67	0
n=1000		N_2	50	20/20	3.1	0
m=49629	21*	N_3	100	20/20	3.8	0
k^*=20		N_4	50	20/20	3.3	0
		N_5	**50**	**20/20**	**3.08**	**0**
		N_1	150	20/20	1546.4	0
	88	N_2	50	20/20	1322.9	0
		N_3	50	20/20	2253.4	0
		N_4	**100**	**20/20**	**852.6**	**0**
		N_5	50	20/20	1116.1	0
DSJC1000.5		N_1	150	0/10	-	5.5
n=1000		**N_2**	**100**	**2/10**	**4998.6**	**1.7**
m=499652	86	N_3	100	0/10	-	12.7
k^*=83		N_4	50	0/10	-	19.3
		N_5	175	2/10	27,071.2	4.8
le450_15c		N_1	25	20/20	0.6	0
n=450		N_2	25	20/20	0.6	0
m=16680	15*	N_3	25	20/20	0.5	0
k^*=**15**		N_4	25	20/20	0.4	0
		N_5	**25**	**20/20**	**0.3**	**0**

Instance	k	N_i	MaxN.	Succ.	Time	Avg
le450_15d		N_1	150	20/20	2.9	0
n=450		N_2	50	20/20	6.4	0
m=16750	15*	**N_3**	**50**	**20/20**	**1.3**	**0**
k^*=**15**		N_4	150	19/20	6.7	0.05
		N_5	50	20/20	3.2	0
le450_25c		N_1	100	7/25	19.9	0.8
n=450	26*	**N_2**	**100**	**25/25**	**10**	**0**
m=17343		N_3	50	18/25	8	0.28
k^*=**25**		N_4	150	18/25	12	0.32
		N_5	100	12/25	19.4	0.68
le450_25d		N_1	100	11/20	61	0.56
n=450	26*	**N_2**	**100**	**20/20**	**11.3**	**0**
m=17425		N_3	150	13/20	6.5	0.52
k^*=**25**		N_4	100	17/20	9	0.4
		N_5	150	7/20	17.2	0.88
flat300_26_0		N_1	50	20/20	4.6	0
n=300		N_2	50	20/20	4.2	0
m=21633	26	N_3	50	20/20	5	0
k^*=**26**		**N_4**	**50**	**20/20**	**3.2**	**0**
		N_5	50	20/20	10.3	0
		N_1	150	15/20	192	0.25
	31*	N_2	150	17/20	160.9	0.15
		N_3	100	17/20	127.8	0.15
		N_4	175	17/20	239.1	0.15
		N_5	**200**	**20/20**	**185**	**0**
Flat300_28_0		N_1	150	10/20	1399.43	1.7
n=300	30	**N_2**	**100**	**10/20**	**1273.6**	**2**
m=21695		N_3	100	6/20	1008.4	2.45
k^*=**28**		N_4	150	9/20	1932.9	2
		N_5	150	8/20	1344.8	2.2
		N_1	50	2/20	547.3	9.35
	29	N_2	200	4/20	2075.8	8.3
		N_3	150	2/20	1876.4	9.35
		N_4	**200**	**4/20**	**1612.7**	**8.2**
		N_5	200	3/20	1190.9	8.75
flat1000_50_0		N_1	50	0/20	2379.8	-
n=1000		N_2	50	0/20	2305.1	-
m=224874	50*	N_3	50	0/20	2226.7	-
k^*=**50**		N_4	50	0/20	1977.4	-
		N_5	**50**	**20/20**	**2858.1**	**0**
flat1000_60_0		N_1	50	0/20	-	-
n=1000		N_2	50	0/20	-	-
m=245830	60	N_3	50	0/20	-	-
k^*=**60**		N_4	50	0/20	-	-
		N_5	**50**	**20/20**	**13,854**	**0**
		N_1	100	20/20	1359.2	0
	87	N_2	50	20/20	1780.5	0
		N_3	150	20/20	3722.4	0
flat1000_76_0		**N_4**	**50**	**20/20**	**859.9**	**0**
n=1000		N_5	50	20/20	1204.7	0
m=246708		**N_1**	**100**	**17/20**	**5750.6**	**0.15**
k^*=82	86	N_2	100	1/20	29,765	2.8
		N_3	50	0/20	-	5.05
		N_4	150	0/20	-	9.88
		N_5	100	2/20	20,579	1.77
		N_1	50	20/20	1803	0
	225*	**N_2**	**50**	**20/20**	**1546**	**0**
		N_3	100	20/20	5467	0
DSJC1000.9		N_4	50	20/20	2542	0
n=1000		N_5	175	20/20	20,816	0
m=449449		N_1	175	8/10	31,461	0.2
k^*=224	224	N_2	50	5/10	13,384	0.7
		N_3	150	5/10	24,483	0.55
		N_4	50	3/10	20,671	1.2
		N_5	**150**	**9/10**	**32,598**	**0.1**
latin_square_10		N_1	50	0/20	-	3.35
n=900	100	N_2	50	0/20	-	3.35
m=307350		N_3	100	2/20	16,878.5	2.3
k^*=98		N_4	50	0/20	-	3.75
		N_5	**100**	**15/20**	**12,812.1**	**0.3**

TABLE 1 – Comparaison des différents voisinages : `CV` : N_1, `CV_C` : N_2, `CV_S` : N_3, `CV_CS` : N_4 and `C_CC` : N_5. Les cellules marquées par * indiquent les valeurs de k pour lesquelles de courtes exécutions sont effectuées.

considéré les $flat300\_x\_0$, avec $x \in \{20, 26, 28\}$ et les $flat1000\_x\_0$, avec $x \in \{50, 60, 76\}$.
- 4 graphes de Leighton **le**n_x issus de l'ordonnancement, ayant 450 sommets et un nombre chromatique connu et égal à la taille de la plus grande clique. Nous avons considéré ceux avec $x \in \{15, 25\}$ et avec l'extension c ou d (i.e., les plus difficiles).
- un graphe de carré latin (latin_square_10).

5.2 Protocole expérimental

Deux séries de tests ont été réalisées avec différents réglages des paramètres `maxMoves` et `maxIter`, correspondants à des exécutions *courtes* et des exécutions plus *longues*. Les exécutions courtes sont utilisées sur les k-coloration "faciles", alors que des exécutions longues sont utilisées sur les grandes denses, avec des valeurs de k proches ou égales aux meilleures colorations connues :

(i) **exécutions courtes** (`maxIter`=n) : `maxMoves`=4500 pour $n = 450$ et `maxMoves` = 50,000 pour $n \in \{300, 500, 1000\}$;

(ii) **exécutions longues** (`maxIter`=$2 * n$) : `maxMoves` = 50,000 pour $n = 250$ et `maxMoves` = 100,000 pour $n \in \{300, 500\}$;

(iii) **exécutions longues** (`maxIter`=$3 * n$) : `maxMoves`=200,000 pour $n = 1000$.

Mentionnons que dans la littérature sur la coloration de graphe, il est courant de faire tourner un algorithme de coloration de plusieurs heures à plusieurs jours pour tenter de résoudre une instance difficile [2, 18].

Notre objectif est d'obtenir un *réglage "générique"* de ces deux paramètres en s'appuyant sur les caractéristiques générales des instances.

Pour chaque paire (G,k), nous avons lancé 20 (ou 10) exécutions sur un Intel Core 2 DUO à 2GHz et à 2GB de RAM. Nous donnons (au plus) les trois plus petites valeurs de k pour laquelle il existe une k-coloration légale [3]. Pour chaque algorithme, nous rapportons le taux de succès ("succ. runs/total runs"), le temps moyen requis pour trouver une solution ainsi

3. Les meilleurs résultats sont disponibles à l'adresse : http ://www.info.unicaen.fr/∼loudni/Coloring.html.

que le coût moyen sur l'ensemble des exécutions. ID-VNS a été implémenté en C++.

5.3 Comparaison des différents voisinages

Pour évaluer les performances de chacun des voisinages, nous avons exécuté ID-VNS avec une seule structure de voisinage ($i_{max} = 1$), avec nextneigh = first et maxneigh $\in \{50, 100, 150, 175, 200\}$.

Le Tableau 1 présente les différents résultats obtenus. La colonne 1 indique les caractéristiques de chaque graphe : son nom, le nombre de sommets (n), le nombre d'arêtes (m) et la meilleure coloration connue dans la littérature (k^*) (en gras quand elle est prouvée optimale). La colonne 2 indique le nombre de couleurs k utilisé. La colonne 3 désigne les différents voisinages. La colonne 4 indique le meilleur réglage pour maxneigh.

Pour comparer les différents voisinages, nous donnons à chaque N_i un score noté (b-s-w), correspondant au nombre de k-colorations pour lesquels N_i obtient respectivement le meilleur (second meilleur entre parenthèses), le même (100%) et le plus mauvais taux de succès. De ces résultats, nous pouvons dresser les remarques suivantes :

- Les voisinages basés sur la topologie du graphe de contraintes (excepté CV_S) sont nettement plus pertinents. Si l'on compare CV et CV_S, le score est de 5(2)-8-11 en faveur de CV ; le score de CV_S est de 1(5)-8-7. Ceci peut s'expliquer par le fait que CV_S effectue des perturbations seulement sur une partie très limitée du graphe (i.e., les voisins d'un sommet centre) alors que le caractère aléatoire de CV permet une plus grande diversification dans l'étape de perturbation, ce qui contribue à trouver des solutions plus facilement.
- CV_CS (avec un score de 5(6)-8-9) est plus performant que CV_S sur la plupart des graphes (excepté DSJC500.1, DSJR500.5 et latin_square_10). Cette comparaison est très instructive et montre l'importance de considérer des sommets plus liés.
- CV_C (avec un score de 8(5)-8-7) et C_CC (avec un score de 11(2)-8-3) surclassent CV_CS. Les deux voisinages trouvent des solutions avec un taux de succès de 100%, respectivement pour 11 et 16 instances, alors que CV_CS atteint le même taux de succès pour 9 instances. En effet, CV_C et C_CC permettent une diversification plus "agressive" en effectuant des perturbations sur différentes sous-parties connexes du graphe de contraintes.
- Enfin, C_CC est légèrement meilleur que CV_C, mais aucun des deux voisinages domine strictement l'autre. En effet, CV_C obtient de meilleurs résultats que C_CC sur 7 instances : (DSJC500.5, $k = 48, 49$), (DSJC1000.5, $k = 89$), le450_25c&d et (flat300_28_0, $k = 29, 30$), alors que C_CC surclasse CV_C sur 9 instances : DSJC250.5, DSJC500.1, DSJR500, (DSJC1000.9, $k = 224$), flat1000_50&60, (flat300_28_0, $k = 31$) et latin_square_10.

5.4 Performance de ID-VNS par rapport à TABUCOL, ID-Walk et VSS-Col

Nous avons comparé ID-VNS à trois algorithmes de recherche locale :

(i) IDWalk [28] : nous avons utilisé une variante basée sur notre recherche tabou. Nous avons effectué des expérimentations avec nextneigh = first et maxneigh $\in \{50, 100, 150, 175, 200\}$. Pour chaque valeur de k, et chaque essai, IDWalk est exécuté 20 fois avec un nombre maximal d'itérations (maxMoves) égal à $1,500,000$. Si aucune k-coloration n'est trouvée, alors il est exécuté 10 fois avec maxMoves égal à $5,000,000$. Un ensemble de 5 essais par k-coloration a été réalisé. Ceci donne une bonne base pour évaluer ID-VNS.

(ii) Deux versions de TABUCOL avec plusieurs restarts [17] : *Short Tabu* et *Long Tabu*. Le choix de TABUCOL réside dans le fait que ID-VNS utilise une variante proche de TABUCOL comme procédure de recherche locale. En outre, il peut être considéré comme une variante de IDWalk avec nextneigh = best [28].

(iii) VSS-Col qui est l'un des algorithmes de recherche locale les plus performants en coloration de graphe [18]. La raison de comparer ID-VNS à VSS-Col est que les deux méthodes étendent VNS. Toutefois, VSS-Col commute itérativement entre plusieurs espaces de recherche, chacun ayant sa propre fonction objective et son voisinage.

Le tableau 2 compare les performances des quatre méthodes. Les résultats de VSS-Col proviennent de [18] et correspondent à ceux obtenus avec une limite de temps d'une heure sur un Pentium 4 à 2 GHz et à 512 MB de RAM. Les résultats de TABUCOL proviennent de [17]. Pour TABUCOL, nous indiquons uniquement la meilleure valeur de k trouvée, tandis que pour ID-Walk, nous donnons aussi le taux de succès (entre parenthèses), le temps moyen en secondes et le coût moyen sur les cinq essais. Les trois dernières lignes donnent un aperçu des comparaisons. Les lignes *better*, *equal* et *worse* donnent le nombre de graphes pour lesquels notre méthode obtient respectivement une meilleure, égale et mauvaise coloration que les autres algorithmes.

De ces résultats, on observe que ID-VNS domine nettement les deux versions de TABUCOL ainsi que ID-Walk, particulièrement sur les graphes de grande taille. En effet, ID-VNS trouve de meilleures colorations que ID-Walk sur 10 graphes (DSJC500.5, DSJC500.9, DSJR500, DSJC1000.5, DSJC1000.9, flat1000 et flat300_28_0). Pour les deux graphes flat1000_50_0 et flat1000_60_0, ID-Walk n'a pas été en mesure de trouver une coloration légale, même pour des valeurs élevées de k. Sur les neuf graphes restants, ID-VNS et ID-Walk trouvent des solutions de même qualité, mais ID-VNS obtient un meilleur taux de succès.

Lorsqu'on compare avec les résultats de VSS-Col, ID-VNS obtient de meilleures solutions sur trois graphes (DSJR500.5, DSJC1000.5 et flat1000_76_0), avec des colorations utilisant respectivement 2, 2 et 1 couleurs en moins. Notons toutefois que ID-VNS n'a pas été capable de trouver une coloration légale avec 20 couleurs pour le graphe DSJC1000.1. Les deux méthodes obtiennent des colorations légales avec le même nombre de couleurs sur 12 graphes. Cependant, si on compare les taux de succès, ID-VNS est bien plus performant que VSS-Col sur trois graphes (DSJC500.9, DSJC1000.9 et flat300_28_0). Les deux méthodes obtiennent des colorations légales avec le même nombre de couleurs et le même taux de succès sur 9 graphes, mais VSS-Col est généralement plus rapide, sauf pour le450_15c et le450_15d, où ID-VNS obtient une coloration légale avec un nombre de couleurs égale au

Instance	k^*	k	ID-VNS		VSS-Col		IDWalk			ShortTabu	LongTabu
			Succ.	Time	Succ.	Time	Best k	Time	AVG	Best k	Best k
DSJC250.1	8	8	20/20	0	-	-	8 (5)	0	0	8	8
DSJC250.5	28	28	20/20	76	-	-	28 (1)	1241	0.8	30	29
DSJC250.9	72	72	20/20	8	-	-	72 (5)	15	0	73	72
DSJC500.1	12	12	20/20	121	10/10	97	12 (5)	1465	0	13	13
DSJC500.5	48	48	2/10	4762	**3/10**	1331	50 (3)	2378	0.4	53	50
		49	9/20	186	**10/10**	162					
DSJC500.9	126	126	18/20	2451	8/10	1686	127 (1)	3435	1	133	130
		127	20/20	153	10/10	169					
DSJR500.1c	85	85	10/10	1713	9/10	736	85 (0)	-	1.4	87	86
DSJR500.5	122	124	**1/10**	297	0/10	-	125 (0)	-	3.4	128	128
		125	**9/10**	2702	0/10	-					
DSJC1000.1	20	20	-	-	**3/10**	2396	21 (5)	4	0	22	22
		21	20/20	3	10/10	11					
DSJC1000.5	83	86	**2/10**	4998	-	-	90 (1)	2711	1.2	95	89
		88	**20/20**	852	8/10	2028					
		89	20/20	1608	10/10	820					
DSJC1000.9	224	224	**9/10**	32,598	1/10	3326	228 (1)	5707	1.2	248	245
		225	**20/20**	1546	5/10	1484					
		226	20/20	2227	10/10	1751					
le450_15c	15	15	20/20	0	10/10	6	15 (5)	3	0	15	15
le450_15d	15	15	20/20	1	10/10	44	15 (5)	5	0	22	16
le450_25c	25	26	25/25	10	10/10	1	26 (3)	6	0.4	27	26
le450_25d	25	26	25/25	11	10/10	1	26 (1)	279	0.8	27	27
flat300_28_0	28	29	**4/20**	1612	1/10	867	31 (1)	486	1.4	-	-
		30	**10/20**	1273	2/10	2666					
		31	20/20	185	10/10	39					
flat1000_50_0	50	50	20/20	2858	10/10	318	60 (0)	-	484.4	-	-
flat1000_60_0	60	60	20/20	13,854	10/10	694	70 (0)	-	223.8	-	-
flat1000_76_0	82	86	**17/20**	5750	-	-	90 (5)	1190	0	-	-
		87	**20/20**	859	4/6	1689					
		88			10/10	1155					
Better					**3**		**10**			**13**	**12**
Equal					12		9			2	3
Worse					**1**		**0**			**0**	**0**

TABLE 2 – Comparaison entre TABUCOL, IDWalk et VSS-Col.

Instance	k^*	MOR	HCA		FOO-P.COL		DYN-P.COL		MMT	ID-VNS	
		k_{best}	Succ.	k	Succ.	k	Succ.	k	k_{best}	Succ.	k
DSJC125.1	5								**5**	20/20	**5**
DSJC125.5	17	**17**							**17**	20/20	**17**
DSJC125.9	44								**44**	20/20	**44**
DSJC250.1	8								**8**	20/20	**8**
DSJC250.5	28	**28**	9/10	**28**					**28**	20/20	**28**
DSJC250.9	72								**72**	20/20	**72**
DSJC500.1	12	**12**			23/50	**12**	50/50	**12**	**12**	20/20	**12**
DSJC500.5	48	49	5/10	**48**	50/50	(49)50	1/50	(48)49	**48**	2/10	**48**
DSJC500.9	126	128			48/50	128	1/50	127	127	18/20	**126**
DSJR500.1c	85	**85**			50/50	**85**	3/50	**85**	**85**	10/10	**85**
DSJR500.5	122	123			24/50	128	28/50	(125)126	**122**	1/10	124
DSJC1000.1	20	21	-	**20**	50/50	21	3/50	**20**	**20**	20/20	21
DSJC1000.5	83	88	-	**83**	5/50	89	6/50	89	84	2/10	86
DSJC1000.9	224	226	-	**224**	30/50	228	30/50	(226)228	225	9/10	**224**
le450_15c	15	**15**	6/10	**15**	50/50	**15**	50/50	**15**	**15**	20/20	**15**
le450_15d	15	**15**			50/50	**15**	50/50	**15**	**15**	20/20	**15**
le450_25c	25	**25**	10/10	26	50/50	(26)27	50/50	(25)27	**25**	20/20	26
le450_25d	25	**25**			50/50	(25)27	50/50	(25)27	**25**	20/20	26
flat300_26_0	26	**26**							**26**	20/20	**26**
flat300_28_0	28	31	6/10	31	35/50	**28**	13/50	**28**	31	4/20	29
flat1000_50_0	50	**50**			50/50	**50**	50/50	**50**	**50**	20/20	**50**
flat1000_60_0	60	**60**			50/50	**60**	50/50	**60**	**60**	20/20	**60**
flat1000_76_0	82	89	4/5	**83**	10/50	88	9/50	88	(82) **83**	17/20	86
latin_square	98								101	15/20	**100**
Better		**6**		**1**		**8**		**8**	**4**		
Equal		10		5		7		6	14		
Worse		**3**		**3**		**1**		**2**	**6**		

TABLE 3 – Comparison aux meilleurs résultats sur les graphes les plus étudiés . Entre parenthèses, certains meilleurs résultats obtenus avec une limite de temps de 10h (non pris en compte dans nos comparaisons).

nombre chromatique très rapidement. Par conséquent, ID-VNS peut être considéré comme plus efficace que VSS-Col, même si notre méthode utilise des voisinages simples au sein d'un même espace de recherche.

Nous avons aussi comparé ID-VNS à deux encodages SAT du problème de coloration (*direct* et avec *support* [14]). MiniSAT, aidé par des techniques de cassage de symétrie, n'a pas été capable de résoudre même les instances denses de petites taille ; les résultats ne sont donc pas rapportés.

5.5 Comparaison avec les meilleurs algorithmes

Dans cette section, nous comparons notre méthode aux résultats des meilleurs algorithmes (voir sec. 2) : **trois algorithmes de recherche locale** (MOR [27] et DYN/FOO-P.COL [2]) et **deux algorithmes hybrides évolutionnistes** (HCA [9] et MMT [22]). Toutefois, nous ne rapportons pas les temps CPU des différentes méthodes, car les conditions d'expérimentation ne sont pas équivalentes. En effet, en plus des différentes machines utilisées, certains algorithmes (MOR, HCA et MMT) nécessitent un réglage spécifique de leurs divers paramètres, qui changent d'une instance à une autre. Les résultats sont donnés afin que le lecteur puisse avoir une base de comparaison pour évaluer les performances de ID-VNS.

Si on compare les résultats de ID-VNS à ceux des méthodes de recherche locale, on peut observer que ID-VNS domine nettement ces algorithmes (voir les trois dernières lignes du tableau 3). En effet, ID-VNS obtient de meilleurs résultats pour au moins six graphes, alors que de moins bons résultats sont obtenus pour au plus trois graphes. En outre, pour les graphes DSJC500.9 (resp. DSJC1000.9), ID-VNS et VSS-Col (resp. HCA) sont les seuls algorithmes capables d'atteindre une 126-coloration (resp. 224-coloration). Le détail des comparaisons est donné ci-dessous :

- ID-VNS est meilleur que MOR sur 6 graphes (DSJC500.5, DSJC500.9, DSJC1000.5, DSJC1000.9, flat300_28_0, et flat1000_76_0) et obtient de moins bons résultats sur 3 graphes (DSJR500.5, le450_25c et le450_25d).
- ID-VNS est meilleur que DYN/FOO-P.COL sur 8 graphes (DSJC500.5, DSJC500.9, DSJC1000.5, DSJC1000.9, le450_25c, le450_25d et flat1000_76_0) et obtient de moins bons résultats pour au plus 2 graphes (flat300_28_0 et/ou DSJC1000.1). Pour flat300_28_0, seul P.COL trouve facilement une 28-coloration.
- Si on considère le taux de succès, ID-VNS est meilleur que VSS-Col sur 6 graphes (DSJC500.9, DSJR500.5,DSJC1000.5, DSJC1000.9, flat300_28_0 et flat1000_76_0) et obtient de moins bons résultats sur 2 graphes (DSJC500.5 et DSJC1000.1).

Lorsqu'on se compare aux deux méthodes hybrides, on observe que ID-VNS est très compétitif. En effet, ID-VNS est meilleur que HCA sur flat300_28_0 et moins bon sur 3 graphes (DSJC1000.1, DSJC1000.5 et flat1000_76_0), meilleur que MMT sur 4 graphes (DSJC500.9, DSJC1000.9, flat300_28_0 et latin_square_10) et moins bon sur 6 graphes.

Cependant, ID-VNS doit être considéré comme une méthode de recherche locale simple qui utilise des mécanismes simples (sans aucun réglage spécifique des paramètres) pour échapper aux minima locaux, alors que HCA et MMT sont plus complexes, avec des ingrédients sophistiqués et finement réglés, qui changent d'une instance à une autre.

5.6 Analyse des paramètres de ID-VNS

Le but de cette section est d'analyser l'influence des paramètres les plus importants de ID-VNS : la liste de mouvements candidats (CLS), les paramètres netxneigh, maxneigh, maxIter et maxMoves.

5.6.1 Influence de nextneigh

Le tableau 4 compare les résultats de ID-VNS avec deux réglages du paramètre nextneigh : first et best. L'impact sur les performances de (nextneigh = first) est très significatif, particulièrement sur les graphes le450_25 et DSJC500, pour lesquels plusieurs ordres de grandeur sont obtenus. Les mauvais résultats de ID-VNS(best) sont probablement dues au fait que la sélection du meilleur voisin (i.e., de plus petit coût) parmi les maxneigh candidats rejetés conduit ID-VNS à rester coincé dans un profond optimum local en bouclant sur des zones déjà visitées. Ce surprenant résultat souligne l'importance de ce paramètre, pour lequel une plus grande diversification est obtenue avec nextneigh = first.

5.6.2 Influence de la liste de mouvements candidats

Pour montrer l'importance de la liste de mouvements candidats (CLS), nous avons comparé ID-VNS avec une version de VNS utilisant une recherche tabou n'exploitant aucune CLS (notée VNS/TS). Comme le montre le tableau 4, ID-VNS(first) est clairement le meilleur. À l'inverse, nous pouvons observer que l'impact de la CLS pour nextneigh=best n'est pas systématique. En effet, comparé à VNS/TS, ID-VNS(best) est très performant sur le450_15c&d. Cependant, sur les autres instances, VNS/TS est bien plus efficace. Ces résultats confirment, une fois de plus, l'importance du mécanisme CLS qui offre un moyen simple et efficace pour parvenir à un meilleur compromis intensification/diversification, en choisissant first.

5.6.3 Influence de maxneigh.

Pour mesurer l'impact du paramètre maxneigh, nous déterminons pour chaque valeur de maxneigh $\in \{50, 100, 150.175\}$ et chaque voisinage N_i, le nombre k-colorations pour lesquels maxneigh obtient respectivement le meilleur (b), second meilleur (entre parenthèses), le même (s) et le plus mauvais taux de succès (w). Le tableau 5 présente le résumé des comparaisons. Comme on pouvait s'y attendre, la valeur optimale de maxneigh dépend de l'instance, ainsi que du voisinage utilisé. Sur les instances "faciles" (i.e., DSJC250.9, le450_15c et le450_15d), l'impact sur les performance (i.e., taux de succès) de maxneigh semble négligeable. Toutefois, il est préférable de privilégier de petites valeurs (50 dans notre cas), probablement parce que les grandes valeurs sont plus coûteuses. À l'inverse, sur les instances difficiles, le gain en performance est très clair pour les valeurs élevées de maxneigh, qui favorisent une *forte intensification* dans l'étape de recherche locale, car plus de voisins sont examinés. Notons toutefois que maxneigh = 100 offre un comportement plus stable pour les cinq voisinages.

Instance	k	N_i	ID-VNS (first)			ID-VNS (best)			VNS/TS		
			Succ.	Time	Avg.	Succ.	Time	Avg.	Succ.	Time	Avg.
le450_15c	15	N_1	20/20	0.6	0	16/20	15	0.9	0/20	-	15.5(7)
		N_2	20/20	0.6	0	10/20	9.1	2.15	5/20	286	7.5
		N_3	20/20	0.5	0	4/20	12.9	17.05	1/20	670	11.5
		N_4	20/20	0.4	0	11/20	11.8	1.2	11/20	377.7	5.3
		N_5	20/20	0.3	0	7/20	17.4	3.1	0/20	-	24.7(2)
le450_15d	15	N_1	20/20	2.9	0	12/20	27.2	1.35	0/20	-	16.1(6)
		N_2	20/20	6.4	0	11/20	76.4	1.05	2/20	468.5	11.7
		N_3	20/20	1.3	0	3/20	99.4	5.7	0/20	-	15.6(4)
		N_4	19/20	6.7	0.05	10/20	64.5	0.95	6/20	447	7.74
		N_5	20/20	3.2	0	5/20	43.6	2.7	0/20	-	31.5(13)
le450_25c	26	N_1	7/25	19.8	0.8	0/20	-	5.6(4)	8/20	246.2	2.05
		N_2	25/25	10	0	0/20	-	6.1(4)	5/20	295.6	2.2
		N_3	18/25	8	0.28	0/20	-	4.4(3)	3/20	330.6	2.8
		N_4	18/25	12	0.32	0/20	-	7(4)	6/20	279.8	2
		N_5	12/25	19.4	0.68	0/20	-	6.6(5)	0/20	-	6.6(2)
le450_25d	26	N_1	11/20	61	0.56	0/20	-	4.95(3)	2/20	198	3
		N_2	20/20	11.3	0	0/20	-	5.4(3)	7/20	233	2.2
		N_3	13/20	6.5	0.52	0/20	-	4.9(3)	6/20	107.7	2.3
		N_4	17/20	9	0.4	0/20	-	6.8(6)	10/20	365.5	1.8
		N_5	7/20	17.2	0.88	0/20	-	6.6(5)	2/20	79.5	5.6
DSJC250.5	28	N_1	18/20	113.5	0.1	0/20	-	4.05(2)	2/20	1946	2
		N_2	18/20	148.1	0.1	0/20	-	3.9(2)	3/20	1436	2.7
		N_3	15/20	83.2	0.25	0/20	-	4(2)	4/20	1318	2.4
		N_4	20/20	118.9	0	0/20	-	4.15(2)	3/20	1494.3	2.6
		N_5	20/20	76.4	0	0/20	-	5(4)	1/20	149	5.5
DSJC500.9	127	N_1	20/20	153.1	0	0/20	-	3.85(2)	3/20	2015.3	3.35
		N_2	20/20	173.3	0	0/20	-	3.95(2)	1/20	5523	3.85
		N_3	19/20	203.3	0.05	0/20	-	3.7(2)	2/20	7686	3.95
		N_4	20/20	208.5	0	0/20	-	4.2(3)	2/20	11,626	3.45
		N_5	19/20	194.2	0.05	0/20	-	5.05(3)	1/20	687	5.3

TABLE 4 – Influence de la liste des mouvements candidats et du paramètre `nextneigh` sur les performances. Pour les essais infructueux, nous rapportons entre parenthèses la meilleure violation trouvée.

5.6.4 Exécutions courtes vs. Longues

Pour mieux analyser l'influence des exécutions courtes et longues sur notre algorithme, nous avons comparé `ID-VNS` avec les deux réglages sur six instances. Le tableau 6 rapportent les résultats obtenus. `ID-VNS` est capable d'améliorer très significativement les taux de succès obtenus avec les exécutions courtes, en particulier sur trois graphes (DSJC250.5, DSJC250.9 et DSJC500.1). Ce comportement montre l'importance des mécanismes de diversification utilisés (degré de perturbation et le paramètre `nextneigh`), qui forcent `ID-VNS` à découvrir des régions inexplorées de l'espace de recherche. Avec plus de temps de calcul, `ID-VNS` est capable de trouver de nouvelles meilleures solutions. Cette importante caractéristique n'est vérifiée que par peu d'algorithmes existants. Très souvent, ces méthodes se bloquent dans des optima locaux et passent leur temps à ré-explorer les mêmes régions dans une boucle, même si on leur donne plus de temps de calcul.

6 Conclusions

Dans ce paper, nous avons proposé une nouvelle extension de l'algorithme de base VNS, appelé `ID-VNS`, qui intègre des mécanismes simples pour réaliser un contrôle efficace de balance diversification/intensification au sein de VNS. Pour atteindre cet objectif, nous avons défini tout d'abord une mesure de diversité, appelée *degré de perturbation*, pour contrôler la quantité de changements nécessaires pour transformer une solution en une autre. Puis, nous avons utilisé une *une liste de mouvements candidats* pour contrôler l'effort d'exploration des voisinages par la recherche locale. Ensuite, pour le problème de coloration de graphe, nous avons défini des structures de voisinages génériques qui bénéficient à la fois des avantages des conflits et de la topologie du graphe de contraintes. Les résultats montrent que les voisinages les plus efficaces sont ceux qui effectuent une diversification "plus agressive".

Des expérimentations réalisées sur les instances difficiles de DIMACS [29] montrent que notre méthode domine nettement les meilleurs algorithmes de recherche locale de coloration de graphe. En outre, malgré sa simplicité, `ID-VNS` semble être très compétitive avec les approches hybrides évolutionnistes, qui sont beaucoup plus complexes. Une première analyse des paramètres de `ID-VNS`, montre l'importance de la CLS ainsi que du paramètre `nextneigh` pour atteindre un meilleur compromis entre l'intensification et diversification.

Comme travaux futurs, doter `ID-VNS` de mécanismes pour adapter automatiquement le choix du voisinage lors de la recherche et aussi d'étudier l'impact du degré de perturbation sur le compromis intensification/diversification.

Références

[1] Cédric Avanthay, A. Hertz, and N. Zufferey. A variable neighborhood search for graph coloring. *European Journal of Operational Research*, 151(2) :379–388, 2003.

[2] I. Blöchliger and N. Zufferey. A graph coloring heuristic using partial solutions and a reactive tabu scheme. *Computers & OR*, 35(3) :960–975, 2008.

[3] J.R. Brown. Chromatic scheduling and the chromatic number problem. *Management Science*, 19(4) :456–463, 1972.

		maxneigh											
		50			100			150			175		
		b	s	w	b	s	w	b	s	w	b	s	w
Hard instances	N_1	3(5)	5	9	**5(3)**	5	**8**	**8(2)**	5	**8**	3(3)	5	10
	N_2	3(4)	5	12	**6(5)**	5	**8**	3(3)	5	10	5(2)	5	12
	N_3	6(7)	1	11	**9(3)**	1	**10**	**7(5)**	1	**11**	5(9)	1	8
	N_4	3(4)	6	12	5(4)	6	9	4(4)	6	11	**7(4)**	6	**7**
	N_5	5(5)	8	8	**5(7)**	8	**4**	6(4)	8	8	7(2)	8	6
Easy instances	N_1	0	4	0	0	4	0	0	4	0	0	4	0
	N_2	**1**	2	**1**	0(1)	3	0	0	3	0	0	3	1
	N_3	**1**	3	**0**	0(1)	3	0	0	3	0	0	3	1
	N_4	0	3	1	0(1)	2	1	**1**	3	**0**	**1**	3	**0**
	N_5	0	4	0	0	4	0	0	4	0	0	4	0

TABLE 5 – Influence du paramètre `maxneigh`.

Instance	k	N_i	MaxN.	Succ.	Time	Succ.Iter.	Avg
DSJC250.5		N_1	200	18/20	113.5	176	0.1
n=250		N_2	175	18/20	148.1	246	0.1
m=15668	28	N_3	100	15/20	83.2	178	0.25
k^*=28		N_4	150	20/20	118.9	190	0
		N_5	**150**	**20/20**	**76.4**	**138**	**0**
DSJC250.9		**N_1**	50	20/20	8.5	19	0
n=250		N_2	100	20/20	9.8	19	0
m=27897	72	N_3	150	20/20	9.9	15	0
k^*=72		**N_4**	**50**	**20/20**	**8.4**	**20**	**0**
		N_5	50	20/20	10.86	26	0
DSJC500.1		N_1	175	17/20	142.4	136	0.15
n=500		N_2	175	17/20	448.9	429	0.2
m=24916	12	N_3	175	15/20	325.7	318	0.4
k^*=12		N_4	100	13/20	303.4	372	0.5
		N_5	**200**	**20/20**	**121.9**	**102**	**0**
DSJC500.5		**N_1**	**100**	**16/20**	**965.4**	**719**	**0.3**
n=500		N_2	175	14/20	889	547	0.35
m=125248	49	N_3	100	11/20	1140.2	862	0.5
k^*=48		N_4	200	16/20	1778.6	700	0.25
		N_5	200	12/20	1894.3	704	0.45

Instance	k	N_i.	MaxN.	Succ.	Time	Succ.Iter.	Avg
DSJC250.5		**N_1**	**50**	**8/20**	**10.46**	**539**	**0.7**
n=250		N_2	100	8/20	11.4	519	1.1
m=15668	28	N_3	175	1/20	3.8	135	2.4
k^*=28		N_4	175	2/20	13.9	494	2.5
		N_5	200	7/20	15.9	458	0.85
DSJC250.9		**N_1**	**200**	**19/20**	**6.27**	**163**	**0.05**
n=250		N_2	50	15/20	5.4	262	0.25
m=27897	72	N_3	50	5/20	3.5	172	1.35
k^*=72		N_4	50	5/20	2.7	133	1.15
		N_5	175	14/20	7.8	232	0.3
DSJC500.1		N_1	150	7/20	88.5	182	1.05
n=500		N_2	50	7/20	100.8	338	1.35
m=24916	12	N_3	100	3/20	164.2	424	1.45
k^*=12		N_4	50	6/20	88.7	303	1.40
		N_5	**50**	**18/20**	**78.4**	**233**	**0.1**
DSJC500.5		**N_1**	**100**	**9/20**	**186.2**	**278**	**0.55**
n=500		N_2	50	5/20	270.1	519	1.15
m=125248	49	N_3	100	7/20	249.4	396	0.75
k^*=48		N_4	100	7/20	246.2	376	1.05
		N_5	150	6/20	494.6	660	0.95

TABLE 6 – Comparaison entre exécutions courtes (à droite) et exécutions longues (à gauche) pour `ID-VNS`.

[4] E. K. Burke, J. Marecek, A. J. Parkes, and H. Rudová. A supernodal formulation of vertex colouring with applications in course timetabling. *Annals OR*, 179(1) :105–130, 2010.

[5] C.Desrosiers, P.Galinier, and A. Hertz. Efficient algorithms for finding critical subgraphs. *Disc. Appl. Math.*, 156(2) :244–266, 2008.

[6] G. J. Chaitin. Register allocation & spilling via graph coloring. In *SIGPLAN Symp. on Compiler Construction*, pages 98–105, 1982.

[7] Dominique de Werra. An introduction to timetabling. *European Journal of Operational Research*, 19(2) :151–162, 1985.

[8] C. Fleurent and J.A. Ferland. Genetic and hybrid algorithms for graph coloring. *Annals of Oper. Research*, 63(3) :437–461, 1996.

[9] P. Galinier and J.-K. Hao. Hybrid evolutionary algorithms for graph coloring. *J. Comb. Optim.*, 3(4) :379–397, 1999.

[10] P. Galinier and A. Hertz. A survey of local search methods for graph coloring. *Computers & OR*, 33 :2547–2562, 2006.

[11] M. Gamache, A. Hertz, and J. O. Ouellet. A graph coloring model for a feasibility problem in monthly crew scheduling with preferential bidding. *Computers & OR*, 34(8) :2384–2395, 2007.

[12] A. Gamst. Some lower bounds for a class of frequency assignment problems. *IEEE Trans. on Vehicular Technology*, 35 :8–14, 1986.

[13] M. R. Garey and D. S. Johnson. *Computers and Intractability : A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[14] Ian P. Gent. Arc consistency in SAT. In *ECAI*, pages 121–125, 2002.

[15] F. Glover and M. Laguna. *Tabu Search*. Kluwer, 1997.

[16] F. Herrmann and A. Hertz. Finding the chromatic number by means of critical graphs. *ACM J. of Exp. Algorithm.*, 7 :10, 2002.

[17] A. Hertz and D. de Werra. Using tabu search techniques for graph coloring. *Computing*, 39(4) :345–351, 1987.

[18] A. Hertz, M. Plumettaz, and N. Zufferey. Variable space search for graph coloring. *Disc. Appl. Math.*, 156(13) :2551–2560, 2008.

[19] F.T. Leighton. A graph coloring algorithm for large scheduling problems. *J. Res. Nat. Bureau of Standards*, 84(6) :489–503, 1979.

[20] A. Linhares and H. Yanasse. Search intensity versus search diversity : a false trade off ? *Appl. Intell.*, 32(3) :279–291, 2010.

[21] S. Loudni, P. Boizumault, and N. Levasseur. Advanced generic neighborhood heuristics for VNS. *Eng. Appl. of AI*, 23(5) :736–764, 2010.

[22] E. Malaguti, M. Monaci, and P. Toth. A metaheuristic approach for the vertex coloring problem. *INFORMS Journal on Computing*, 20(2) :302–316, 2008.

[23] E. Malaguti and P. Toth. A survey on vertex coloring problems. *Intl. Trans. in Op. Res.*, 17(1) :1475–3995, 2010.

[24] A. Mehrotra and M. Trick. A column generation approach for graph coloring. *INFORMS Journal on Computing*, 8(4) :344–354, 1996.

[25] I. Méndez-Díaz and P. Zabala. A branch-and-cut algorithm for graph coloring. *Disc. Appl. Math.*, 154(5) :826–847, 2006.

[26] N. Mladenović and P. Hansen. Variable neighborhood search. *Computers And Operations Research*, 24 :1097–1100, 1997.

[27] C. Morgenstern. Distributed coloration neighborhood search. volume 26 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 335–357. American Mathematical Society, Providence, RI, USA, 1996.

[28] B. Neveu, G. Trombettoni, and Fred Glover. ID Walk : A candidate list strategy with a simple diversification device. In *Proceedings of CP'04, LNCS 3258*, pages 423–437, 2004.

[29] M. Trick. Computational symposium : Graph coloring and its generalizations. In *Cornell University, Ithaca, NY*, 2002. http ://mat.gsia.cmu.edu/COLOR02/.

De la complexité des problèmes de contraintes

Florent R. Madelaine

Université d'Auvergne * LIMOS †

Florent.Madelaine@u-Clermont1.fr

Résumé

La *conjecture de la dichotomie* postule qu'un problème de satisfaction de contraintes est soit facile (dans P), soit difficile (NP-complet). Cette conjecture, motivée par Feder et Vardi il y a 20 ans, reste ouverte malgré les efforts importants d'une communauté internationale regroupant des chercheurs issus d'horizons très variés. Nous présentons ici un bref aperçu des résultats obtenus et des techniques utilisées pour étudier cette question centrale en informatique théorique afin d'illustrer cette interdisciplinarité qui mêle algèbre, combinatoire, complexité et logique.

Abstract

The *dichotomy conjecture* asserts that a constraint satisfaction problem is either tractable (in P) or hard (NP-complete). This conjecture, motivated by Feder and Vardi 20 years ago, remains open, despite important efforts from a growing international community that regroups researchers from diverse horizons. We present here a brief overview in French of results and techniques used in attempts to settle this conjecture, in order to illustrate this multidisciplinarity which combines Algebra, Combinatorics, Complexity and Logic and makes this a central question in Theoretical Computer Science.

1 Introduction

Il est bien connu que les problèmes de contraintes sont très généraux et permettent de modéliser naturellement de nombreux problèmes combinatoires et industriels difficiles. La généralité qu'offre ce cadre de travail et l'existence de méthodes génériques comme la propagation de contraintes est la motivation première de la communauté IA qui s'attache à développer des solveurs efficients tant au niveau de la modélisation que du calcul d'une solution. Ce qui est peut-être moins connu, et qui indique la robustesse des problèmes de contraintes, est leur ubiquité : ils sont étudiés sous d'autres noms en *bases de données* (inclusion de requêtes conjonctives), en *combinatoire et théorie des graphes* (problème d'existence d'homomorphisme, coloriage de graphes) et en *logique* (évaluation de formules primitives positives). La complexité de ce(s) problème(s), en particulier les cas pour lesquels il existe un algorithme polynomial [1] semblent s'expliquer soit de manière combinatoire, soit ce qui est peut-être plus surprenant par des *propriétés algébriques*.

La motivation ici n'est pas de présenter les détails techniques mais de citer certain résultats théoriques importants, d'en expliquer l'intuition et de les remettre dans leur contexte en espérant que le lecteur se tournera vers des *survey* récents en anglais (le livre [14] en regroupe plusieurs). Nous commencerons en §2 par des exemples et par deux théorèmes de dichotomie historiquement importants, celui de Schaefer qui concerne le cas Booléen et celui de Hell et Nešetřil qui concerne les graphes non-orientés. Ces théorèmes ont conduit Feder et Vardi à avancer la conjecture de la dichotomie.

Au delà de l'intérêt évident d'une caractérisation des cas polynomiaux, cette conjecture est motivée par des résultats de *complexité structurelle* puisque la classe des problèmes de contraintes serait « la plus large » pour laquelle on observerait ce phénomène de dichotomie. On abordera cet aspect en §3.

Deux approches fondamentalement différentes permettent de restreindre le problème pour obtenir des classes polynomiales. Dans le premier cas, la notion de décomposition arborescente bornée du graphe des contraintes, bien connue depuis les travaux fondateurs de Freuder [21] est centrale et on verra en §4 qu'il existe un algorithme polynomial même lorsqu'il n'existe pas directement de telle décomposition. Dans le second

*Clermont Université, Université d'Auvergne, LIMOS, BP 10448, F-63000 Clermont-Ferrand.

†CNRS, UMR 6158, Laboratoire d'Informatique, de Modélisation et d'Optimisation des Systèmes, F-63173 Aubière.

1. pour le problème de décision (calculer, compter ou énumérer les solutions sont aussi des questions importantes mais on ne les abordera pas ici par manque de place).

cas, on restreint le langage de contraintes, et on verra en §5 que l'algèbre joue un rôle prépondérant et permet de caractériser par exemple le fait que pouvoir établir un certain niveau de cohérence suffit pour pouvoir déterminer si une solution existe. La *conjecture de la dichotomie* concerne en fait ce second cas et reste ouverte contrairement au premier cas qui est essentiellement complètement classifié. Finalement, nous nous tournerons en §6 vers des variantes, comme les problèmes de contraintes quantifiées, pour lesquels la méthode algébrique peut être adaptée avec succès.

2 Préliminaires

Une instance du *problème de satisfaction de contraintes* est traditionnellement définie comme un triplet $(\mathsf{Var}, \mathsf{Dom}, \mathcal{C})$, où Var est un ensemble de variables, Dom est un ensemble fini de valeurs et $\mathcal{C}$ est un ensemble de contraintes. Chaque contrainte est de la forme $(v_{i_1}, \ldots, v_{i_r}, R)$, où r est l'arité de la contrainte et $R \subseteq \mathsf{Dom}^r$ une relation spécifiant tous les r-uplets de valeurs admissibles simultanément par les variables $v_{i_1}, \ldots, v_{i_r}$. Une *solution* de cette instance est une application qui associe une valeur à chaque variable de sorte que chaque contrainte soit satisfaite, à savoir qu'à $v_{i_1}, \ldots, v_{i_r}$ correspond un r-uplet admissible.

Exemple. Le problème de satisfiabilité propositionelle (Sat) peut facilement se coder comme un problème de contraintes. On parle alors de *Generalized Satisfiability* puisque les entrées des deux problèmes sont légèrement différentes. En effet, on aura un domaine booléen $\mathsf{Dom} = \{0, 1\}$, et à la clause $x \vee y \vee \bar{z}$ de Sat on fait correspondre la contrainte ternaire portant sur les variables x, y, z et de relation $\{0, 1\}^3 \setminus \{(0, 0, 1)\}$.

L'exemple précédent montre que le problème de contraintes est NP-complet. La généralité de ce cadre de travail n'est donc pas surprenante puisque si par *modéliser par des contraintes* on entend *réduction polynomiale au problème de satisfaction de contraintes*, tout problème Ω de NP est modélisable en ce sens.

Remarque. Mais est-ce-vraiment la bonne notion de modélisation ? Probablement pas puisque si le problème de départ Ω était facile on voudrait le réduire à une restriction Ω' du problème de contraintes qui soit aussi facile. Nous aborderons plus en détail cette aspect en §3.

On reformule souvent le problème de contraintes en tant que *problème d'homomorphisme* puisqu'on peut naturellement séparer une instance en deux structures similaires – la première $\mathcal{A}$ de domaine Var correspond essentiellement au graphe des contraintes, la seconde $\mathcal{B}$ de domaine Dom regroupe les relations listant les r-uplets admissibles – et qu'une solution correspond exactement à un homomorphisme . Un *homomorphisme* de graphe est une application de sommets à sommets qui doit nécessairement envoyer une arête sur une arête. Le cas des graphes orientés est similaire sauf qu'on préservera aussi l'orientation d'un arc. Nous allons expliciter cette reformulation sur un exemple simple ci-dessous. Un exemple plus complexe avec plusieurs contraintes et donc des structures avec plusieurs relations est donné pour le cas de 2-Sat page suivante.

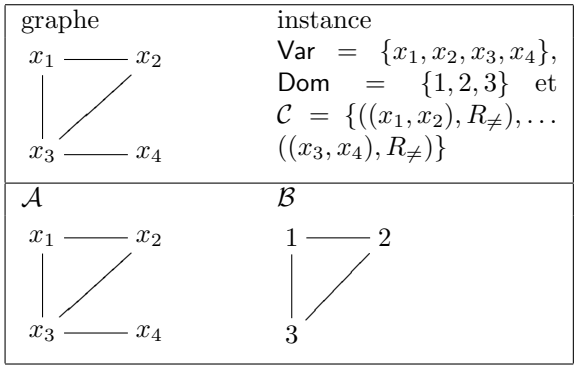


FIGURE 1 – Reformulation de la 3-colorabilité comme problème d'homomorphisme

Exemple. On peut facilement coder le problème de 3-colorabilité d'un graphe $G := (V, E)$ comme problème de contraintes. Les sommets sont les variables, soit $\mathsf{Var} := V$, le domaine correspond aux trois couleurs $\mathsf{Dom} := \{1, 2, 3\}$ et puisque les sommets incidents à une arête doivent prendre une couleur différente, on poste la contrainte $(x_i, x_j, R_{\neq})$ pour chaque arête (x_i, x_j) dans E, où $R_{\neq}$ est la relation binaire $\{1, 2, 3\}^2 \setminus \{(1, 1), (2, 2), (3, 3)\}$ listant les paires de couleurs admissibles.

Sous forme de problème d'homomorphisme, dans notre cas simplifié $\mathcal{A}$ et $\mathcal{B}$ seront des graphes. Pour un graphe G, on aura $\mathcal{A} := G$ et $\mathcal{B}$ qui sera un triangle K_3 (*cf.* figure 1). Ce problème est NP-complet alors que la 2-colorabilité est polynomiale.

En combinatoire, de nombreux problèmes difficiles deviennent faciles lorsque l'instance est un arbre. C'est le cas pour n'importe quel problème exprimable en logique monadique du second ordre, d'après le célèbre méta-théorème de Courcelle [12]. On verra en § 3 que notre problème s'exprime dans cette logique.

Théorème 1. *Si on se place dans le cas des graphes et qu'on restreint $\mathcal{A}$ comme étant un arbre, le problème d'homorphisme est polynomial.*

Remarque. On peut remplacer *graphe* par *structure* et *arbre* par *structure arborescente bornée*. Le résultat de Courcelle est très général et reste trop théorique

avec des constantes impratiquables (des tours d'exponentielle liées à la construction d'automates d'arbres). Par contre, pour le cas restreint qui nous intéresse, il existe un algorithme pratique dans l'esprit des travaux de Freuder [21] : on résoud localement en progressant du bas vers le haut dans l'arbre. Il s'agit d'un algorithme de type *bucket elimination* [17].

Le problème du H-*coloring* généralise le problème de coloriage de graphe. Il s'agit de savoir étant donné un graphe G si il existe un homomorphisme de G dans H (le k-coloriage correspond au choix d'une k-clique pour H) [2]. Si le graphe H est biparti, alors ce graphe H est 2-colorable. C'est-à-dire qu'il existe un homomorphisme de H vers K_2 (le graphe qui consiste en une seule arête). Si le graphe H contient au moins une arête, alors il existe un homomorphisme qui envoie l'arête de K_2 sur cette arête de H. Ainsi H et K_2 sont dit *homomorphiquement équivalent* puisque tout graphe G qui admet un homomorphisme vers l'un en admet un vers l'autre (car deux homomorphismes se composent naturellement pour donner un homomorphisme). Notez de plus que K_2 est le plus petit sous-graphe de H qui lui est homomorphiquement équivalent. On dira dans ce cas que K_2 est le *core* [3] de H. Du point de vue du problème de décision, le H-coloring et le 2-coloriage sont exactement le même problème, lorsque H est biparti et contient au moins une arête. Le problème du H-coloring est donc polynomial si H est biparti. Une analyse combinatoire subtile permet de montrer qu'il s'agit là des seuls cas faciles. À partir des cas difficiles connus comme les cliques de taille 3 ou plus, la preuve procède par étapes successives afin d'augmenter cette classe progressivement. La preuve se termine puisqu'on impose tellement de conditions sur les graphes restants qu'on fini par montrer une contradiction. Il s'agit d'une preuve qui relève de la théorie des graphe extrémaux.

Théorème 2 (Hell et Nešetřil 1990 [24])**.** *Le H-coloring est polynomial si H est biparti et NP-complet sinon.*

En général on considère des structures relationnelles quelconques et non pas juste des graphes. On rappelle qu'une *structure relationnelle* $\mathcal{B}$ consiste en un ensemble B (toujours fini dans cette note) et une liste finie de relations sur B. La notion d'homomorphisme entre deux structures similaires généralise naturellement le cas des graphes : il s'agit d'une fonction qui envoie chaque tuple de chaque relation de la première structure sur un tuple de la relation correspondante dans la seconde structure.

2. Ce H correspond à notre $\mathcal{B}$ partout ailleurs, nous gardons le H ici pour préserver la nomenclature utilisée en combinatoire.

3. En anglais, le terme est *core*.

Exemples. On peu coder 2-Sat comme suit. Soit $R^{\mathcal{B}}_{ab} = \{0,1\}^2 \setminus \{(a,b)\}$ et $\mathcal{B} = (\{0,1\}; R^{\mathcal{B}}_{00}, R^{\mathcal{B}}_{01}, R^{\mathcal{B}}_{11})$. Une instance $F = (\neg x \vee \neg z) \wedge (x \vee y) \wedge (y \vee \neg z) \wedge (u \vee x) \wedge (x \vee \neg u)$ correspond à une structure $\mathcal{A}$ de domaine $\{x, y, z, u\}$ et de relations $R^{\mathcal{A}}_{00} = \{(x, y), (u, x)\}$, $R^{\mathcal{A}}_{01} = \{(y, z), (x, u),\}$ et $R^{\mathcal{A}}_{11} = \{(x, z)\}$. Notez la correspondance naturelle entre homomorphisme de $\mathcal{A}$ à $\mathcal{B}$ et assignation valide de F. Ce problème est polynomial (NL-complet pour être précis).

De même, on peut coder *Horn 3-Sat* comme suit. On dispose d'une relation ternaire pour les clauses de Horn non triviales et de deux relations unaires pour les clauses unitaires. On pose $\mathcal{B} = (\{0,1\}; R; \{0\}, \{1\}\}$ où $R = \{(x, y, z) \mid y \wedge z \rightarrow x\}$. Ce problème est P-complet. Dans ce cas, il est bien connu que les relations de $\mathcal{B}$ sont préservées par l'opération booléenne $\wedge$ et qu'il en est de même pour l'ensemble des solutions d'un problème de type Horn en général. Ceci est la clé de l'approche algébrique que nous expliciterons en §5. On dit que $\wedge$ préserve la relation $R = \{0,1\}^3 \setminus \{(0,1,1)\}$ puisque pour tout choix de 2 triplets de R, par exemple $(1,1,0)$ et $(1,0,1)$ en appliquant $\wedge$ à chaque coordonnée on obtient le triplet $(1,0,0)$ qui appartient lui aussi à R.

Pour 2-Sat, l'opération booléenne ternaire h qui retourne toujours la valeur majoritaire (par exemple $h(0,1,1) = 1$) joue un rôle similaire.

Remarque. Notez que $\mathcal{A}$ peut être une structure quelconque. Ainsi, les exemples ci-dessus restent polynomiaux même si $\mathcal{A}$ est une structure non arborescente analogue à une grille par exemple.

Cette vision des problèmes de contraintes via l'homomorphisme est assez populaire car elle permet de séparer de manière naturelle *deux approches fondamentalement différentes* qui permettent d'obtenir des restrictions polynomiales. Les restrictions sur $\mathcal{A}$ sont combinatoires (lien avec les décompositions arborescentes déjà évoquées) alors que les restrictions sur $\mathcal{B}$ sont régies par les propriétés algébriques de fonctions associées à $\mathcal{B}$, comme pour les deux exemples ci-dessus [4].

Un résultat historiquement très important, qui entre dans ce second cas correspond à la classification de *Generalized Satisfiability*. Nous en esquisserons une preuve en §5.

Théorème 3 (Schaefer 1978 [13])**.** *Si le domaine de $\mathcal{B}$ a deux éléments alors le problème d'homomorphisme est soit polynomial, soit NP-complet. Si les relations de $\mathcal{B}$ sont* 0-valide*,* 1-valide*,* 2-Sat*,* Horn*,* dual-Horn*, ou* affine *alors le problème devient polynomial, sinon il est NP-complet.*

4. Notons que même si c'est plutôt contre-intuitif, le théorème de Hell et Nešetřil s'explique lui aussi algébriquement [4].

À l'instar du problème du H-coloring, pour lequel H est fixé, on fixe fréquemment la structure $\mathcal{B}$ et seule la structure $\mathcal{A}$ est une entrée du problème. On dénote ce problème par $CSP(\mathcal{B})$ et on parle dans ce cas de problème de contrainte *non-uniforme*. On notera par la suite par CSP la classe formée par ces problèmes non-uniformes.

Remarque. Cette vision est quelque peu restrictive et ne permet pas de capturer les *contraintes globales*. Cependant, il est fréquent que les algorithmes « uniformisent ». Nous reviendrons sur cet aspect plus loin.

3 Conjecture de la dichotomie

Un corollaire du théorème de Ladner [27] est que si P est différent de NP, alors il existe des problèmes de complexité intermédiaire *qui ne sont ni dans P, ni NP-complet*. En d'autres termes, on ne devrait pas pouvoir prouver de théorème de dichotomie pour NP.

Dans un article qui a eu beaucoup d'influence [20], Feder et Vardi établissent que trois classes de complexité, qu'on ne définira pas formellement ici, $\mathcal{L}_1$, $\mathcal{L}_2$ et $\mathcal{L}_3$ sont *calculatoirement équivalentes à NP*, c'est-à-dire que pour tout problème Ω de NP, il existe un représentant Ω' dans la classe $\mathcal{L}_i$ telle que Ω se réduit à Ω' en temps polynomial et inversement Ω' se réduit à Ω. Ainsi, structurellement ces trois classes se comportent exactement comme NP et en particulier, par le théorème de Ladner, elles ne peuvent pas avoir de dichotomie (on suppose dorénavant que P est différent de NP).

Ces trois classes de complexité se définissent de manière logique et le fragment immédiatement en dessous de ces trois classes correspond à des problèmes exprimables dans la logique MMSNP, qui est un fragment de la logique monadique du second ordre chère à Courcelle [12]. Cette logique permet d'exprimer des problème combinatoires de la forme suivante : il existe un coloriage de l'entrée qui interdit localement un nombre fini d'obstructions coloriées. En particulier on peut exprimer les problèmes $CSP(\mathcal{B})$, pour tout $\mathcal{B}$.

Exemple. Pour la 3 colorabilité, les obstructions seront simplement les arêtes rouge-rouge, vert-vert et bleue-bleue. La formule de MMSNP sera de la forme suivante : $\exists$ une partition R, V, B des sommets telle que $\forall$ sommets x, y $\neg\big(E(x,y) \wedge R(x) \wedge R(y)\big) \wedge \neg\big(E(x,y) \wedge V(x) \wedge V(y)\big) \wedge \neg\big(E(x,y) \wedge B(x) \wedge B(y)\big)$.

La logique MMSNP ne correspond pas exactement à CSP, car elle est trop générale, mais elle a de nombreux points communs avec CSP.

Théorème 4 (Feder et Vardi 1993 [20], Kun [5])**.**

1. *Tout CSP s'exprime par une formule de MMSNP, mais il existe des formules de MMSNP qui n'expriment pas un problème de CSP.*
2. *Par contre, MMSNP est calculatoirement équivalente à CSP.*

Remarques. La logique MMSNP permet d'exprimer un problème comme « pas de triangle » qui n'est pas un problème de contraintes non-uniforme. Il s'agit en fait d'un problème de contrainte non-uniforme *de domaine infini* au sens de Bodirsky [1]. On peut déterminer de manière effective lorsqu'un problème de MMSNP est un CSP à domaine fini ou bien à domaine infini [32]. Pour MMSNP, l'inclusion de 2 problèmes est décidable mais plus complexe que pour CSP : pour CSP, l'inclusion est un CSP uniforme (NP-complet) alors que pour MMSNP on est au moins au second niveau de la hiérarchie polynomiale [29].

Les résultats de dichotomie historiques de Schaefer, et Hell et Nešetřil et des considérations techniques sur la nature foncièrement différente d'une classe $\mathcal{L}_i$ et de MMSNP ont conduit Feder et Vardi à avancer la conjecture suivante en 1993.

Conjecture de la dichotomie. *Tout problème de contrainte non-uniforme est soit polynomial soit NP-complet.*

Notons que cette conjecture implique que MMSNP a une dichotomie puisque MMSNP est calculatoirement équivalent à CSP. L'inverse est trivialement vrai puisque CSP est une restriction de MMSNP. Puisque, les classes de complexité $\mathcal{L}_i$ « immédiatement au dessus » de MMSNP contiennent des problèmes de complexité intermédiaire et que MMSNP est calculatoirement équivalent à CSP, on peut donc voir CSP comme la plus grande classe qui devrait avoir une dichotomie.

De manière générale, on dira qu'une classe de complexité $\mathcal{C}$ est *dichotomie-complète* si $\mathcal{C}$ est calculatoirement équivalent à CSP, puisque démontrer que $\mathcal{C}$ a une dichotomie impliquerait la conjecture de la dichotomie.

Exemples. Le problème d'homomorphisme pour les graphes orientés est dichotomie-complet [20]. Le problème d'homomorphisme pour les algèbres ayant deux fonctions unaires est dichotomie-complet [19].

Lorsqu'on s'intéresse au cas spécial où la formule MMSNP est du premier ordre, c'est-à-dire que les obstructions ne sont pas coloriées comme dans le cas de l'exemple « pas de triangle », le cas où cette formule

5. L'une des réductions reposait sur un lemme dit de Erdös de nature aléatoire, mais ce lemme a été déterminisé par Kun.

exprime un CSP fini est étudié sous le nom de *paire duale* en combinatoire. Par exemple, si la formule n'a qu'une obstruction qui est un chemin orienté de longueur n, cela revient à avoir un homomorphisme dans un tournoi transitif à n sommets. L'étude des paires duales et de la dualité est une question importante en combinatoire, voir [7] pour un survey.

On peut aussi se poser la question si cette différence d'expressivité entre MMSNP et CSP s'estompe si on restreint la classe des structures considérées. Ceci est le cas lorsque les structures sont de degré borné, sont planaires, ou plus généralement définissables par mineurs interdits [6] : dans ces cas, une formule de MMSNP, qui définit en général un CSP de domaine infini, devient forcément un CSP de domaine fini [28].

4 Graphe des contraintes arborescent

Nous nous tournons dans cette section vers les restrictions dites structurelles, qui concernent uniquement la structure $\mathcal{A}$, la structure $\mathcal{B}$ étant quelconque. Il s'agit de restrictions qui se comportent particulièrement bien et qui impliquent l'existence d'un algorithme polynomial même lorsque $\mathcal{B}$ n'est plus fixe et participe aussi à l'entrée. On dénote ce problème dit uniforme par $CSP(\mathscr{C}, \_)$, pour le cas où l'entrée $\mathcal{A}$ appartient à une classe $\mathscr{C}$ de structure et $\mathcal{B}$ est quelconque.

Question. *Quelles sont les classes $\mathscr{C}$ pour lesquelles $CSP(\mathscr{C}, \_)$ est polynomial ?*

Nous avons évoqué qu'un problème de contrainte est facile à résoudre si l'entrée est un arbre. Si l'entrée n'est pas un arbre mais peut être décrite par une décomposition arborescente de largeur au plus k, où k est une constante, alors le problème peut être résolu en temps polynomial n^k. Par contre cet algorithme nécessite d'avoir la décomposition. Décider si une structure $\mathcal{A}$ a une telle largeur arborescente est NP-complet si k est une donnée du problème. Par contre si la largeur k est fixée, il existe un algorithme linéaire en la taille de $\mathcal{A}$ qui calcule, si c'est possible, une telle décomposition.

On travaille sur des structures et non des graphes. On se ramène à un graphe en considérant « le graphe des contraintes ». Ce graphe est connu sous le nom de graphe de Gaifman (de $\mathcal{A}$) en logique. Ce graphe a le même domaine que $\mathcal{A}$ et on a une arête entre deux sommets si ils participent à un même tuple d'une relation quelconque de $\mathcal{A}$. Ainsi, les sommets de $\mathcal{A}$ participant à un tuple formeront une clique dans ce graphe. La complexité de l'algorithme sera alors $n^{O(k)}$ où k est la largeur arborescente du graphe de Gaifman, le $O(k)$ cachant la complexité du codage d'une structure relationnelle en un graphe. On peut donc prendre pour $\mathscr{C}$ la classe $\mathscr{T}_k$ des structures $\mathcal{A}$ dont le graphe de Gaifman a largeur d'arborescence au plus k.

On peut se demander si cette restriction structurelle est la plus générale possible. En fait, on peut autoriser $\mathcal{A}$ a être non pas une structure de $\mathscr{T}_k$ mais *une structure homomorphiquement équivalente à une structure $\mathcal{A}'$ appartenant à $\mathscr{T}_k$*. On notera cette nouvelle classe par $\mathscr{HT}_k$.

Théorème 5 (Dalmau, Kolaitis, Vardi [16])**.** *Pour tout k fixé, le problème de contrainte uniforme $CSP(\mathscr{HT}_k, \_)$ est polynomial.*

Exemple. On peut par exemple considérer l'ensemble $\mathscr{B}$ des graphes bipartis. Lorsque un graphe biparti a une arête, son cœur est K_2 et sinon un seul sommet c'est-à-dire K_1. Ces deux graphes sont des arbres, donc $\mathscr{B}$ est inclus dans $\mathscr{HT}_1$. Notez que $\mathscr{B}$ contient les grilles qui sont le prototype même du graphe n'ayant pas une largeur arborescente bornée. $CSP(\mathscr{B}, \_)$ est polynomial pour une raison triviale : pour une entrée $(\mathcal{A}, \mathcal{B})$, il suffit de tester si $\mathcal{B}$ contient une arête ou si $\mathcal{A}$ n'en contient pas.

Pourquoi le problème $CSP(\mathscr{HT}_k, \_)$ est-il polynomial ? Puisque $\mathcal{A}$ et $\mathcal{A}'$ sont homomorphiquement équivalents, ces deux structures seront ou bien simultanément acceptées ou bien simultanément rejetées du fait de leur équivalence (on rappelle que la composition de deux homomorphismes est un homomorphisme). Ainsi, il suffit étant donné $\mathcal{A}$ de calculer $\mathcal{A}'$, de calculer une décomposition de $\mathcal{A}'$ et d'utiliser l'algorithme polynomial reposant sur cette décomposition. Le problème avec cet argument c'est que calculer $\mathcal{A}'$ est une question difficile. En effet, étant donné $\mathcal{A}$, déterminer si un tel $\mathcal{A}'$ existe est NP-complet, même lorsque k est fixé. L'algorithme polynomial pour $CSP(\mathscr{HT}_k, \_)$ ne nécessitera pas le calcul d'une décomposition : il repose sur un jeu associé à l'expressivité dans une logique, qui est un fragment de la logique du premier ordre n'ayant que $k+1$ variables.

Dans le cas des arbres, on a $k = 1$. La logique a seulement 2 variables et dans le jeu associé, chaque joueur a deux pions. Le jeux oppose le *saboteur* au *copieur* qui jouent sur deux structures $\mathcal{A}$ et $\mathcal{B}$: le premier essaye d'exhiber des différences entre $\mathcal{A}$ et $\mathcal{B}$ alors que le second essaye de montrer leur similarité. Le saboteur joue sur $\mathcal{A}$ et le copieur sur $\mathcal{B}$.

Au début, le saboteur pose deux pions p_1 et p_2 où il le veut sur deux sommets de la structure $\mathcal{A}$ (les som-

6. Un mineur d'un graphe G est un graphe H obtenu par contraction d'arête(s) et suppression de sommet(s) de G. Le célèbre théorème de Kuratowski caractérise les graphes planaires comme étant les graphes n'ayant ni K_5 ni $K_{3,3}$ comme mineur. La classe des graphes planaires est un exemple de classe définissable par mineurs interdits.

mets pouvant coïncider). Le copieur répond en posant deux pions q_1 et q_2 sur deux sommets de $\mathcal{B}$. Ensuite, à chaque tour suivant, le saboteur ramasse un ou plusieurs pions et les replace sur $\mathcal{A}$ où il le souhaite ; et, le copieur procède de même. Par exemple, si le saboteur déplace son pion p_2, le copieur doit jouer q_2.

Le saboteur gagne la partie si après un tour donné, la fonction qui envoie le sommet de $\mathcal{A}$ sur lequel est placé p_1 sur le sommet de $\mathcal{B}$ associé au premier pion du copieur q_1 et de même pour la paire de sommets correspondant à p_2 et q_2, n'est pas un homomorphisme partiel de $\mathcal{A}$ dans $\mathcal{B}$. Dans le cas des graphes, cela correspond au cas où les pions du saboteur sont adjacents sur $\mathcal{A}$ et ceux du copieur ne le sont pas sur $\mathcal{B}$. Le copieur a une stratégie gagnante si il peut jouer indéfiniment sans perdre.

Si il existe un homomorphisme de $\mathcal{A}$ dans $\mathcal{B}$, alors le copieur peut utiliser cet homomorphisme pour répondre à n'importe quel coup du saboteur et a donc une stratégie gagnante.

Si $\mathcal{A}$ est un chemin et que le copieur a une stratégie gagnante, on peut construire un homomorphisme h de $\mathcal{A}$ dans $\mathcal{B}$. On utilise les deux pions du saboteur pour « marcher » le long du chemin $\mathcal{A}$ en déplaçant alternativement le premier pion puis le second pion. On regarde la réponse du copieur. L'homomorphisme partiel existant à chaque étape construit progressivement l'homomorphisme de $\mathcal{A}$ dans $\mathcal{B}$. L'argument est similaire si $\mathcal{A}$ est un arbre.

On a donc démontré que si $\mathcal{A}$ est un arbre, alors il y a un homomorphisme de $\mathcal{A}$ dans $\mathcal{B}$ si et seulement si le copieur a une stratégie gagnante. Il se trouve qu'on peut déterminer en temps polynomial si c'est le cas. On a donc pour l'instant une preuve alternative du théorème 1 utilisant la logique et les jeux. Si $\mathcal{A}$ n'est pas un arbre mais que $\mathcal{A}'$ est un arbre homomorphiquement équivalent à $\mathcal{A}$, on peut utiliser ces homomorphismes pour montrer que le copieur a une stratégie gagnante si et seulement si il y a un homomorphisme de $\mathcal{A}$ dans $\mathcal{B}$.

Le théorème précédent caractérise précisément les restrictions structurelles qui permettent d'obtenir un algorithme polynomial, puisque Grohe montre l'implication inverse.

Théorème 6 (Dalmau *et. al.*[16], Grohe [22])**.** *Sous une hypothèse de complexité paramétrée*[7]*, pour toute classe de structures $\mathscr{C}$ qui est récursivement énumérable, $CSP(\mathscr{C}, \_)$ est polynomial si et seulement si $\mathscr{C}$ est inclus dans $\mathscr{H}\mathscr{T}_k$, pour k fixé.*

À la lumière de ces résultats, on peut penser que la question des restrictions structurelles est complètement résolue. Ce n'est pas tout à fait le cas puisque le cadre de travail proposé a deux inconvénients. D'une part, il ne permet que de travailler dans le cas où l'arité d'une contrainte est bornée, ce qui est plutôt réducteur si on voit l'importance qu'ont prises les contraintes globales. D'autre part, la polynomialité de $CSP(\mathscr{H}\mathscr{T}_k, \_)$ est en quelque sorte inaccessible puisque le méta-problème qui consiste à décider si $\mathcal{A}$ appartient à $\mathscr{H}\mathscr{T}_k$ est NP-complet même si k est fixé.

Ce domaine d'étude reste donc particulièrement actif. Il y a de nombreux travaux portant sur la décomposition d'hypergraphes plutôt que de graphes afin de pouvoir appréhender le cas des contraintes globales. Ces travaux ont beaucoup de liens avec les bases de données puisque le problème de satisfaction de contraintes d'instance $(\mathcal{A}, \mathcal{B})$ correspond au problème d'évaluation d'une requête conjonctive $\varphi_{\mathcal{B}}$ sur une base de données $\mathcal{A}$. Pour plus de détails, voir par exemple [36].

La notion la plus générale dans le cas des hypergraphes est lié à la notion de *fractional hyper-tree width* proposée par Grohe et Marx [23]. Une classification complète par rapport à la complexité paramétrée a été donné par Marx [35].

5 Approche algébrique

Dans cette section, on considère que la structure $\mathcal{B}$ de domaine Dom et de relations Γ est fixée. On appelle Γ le *langage des contraintes*. On s'intéresse au pouvoir d'expressivité du langage de contrainte Γ. En particulier, on aimerait pouvoir comparer l'expressivité de différents langages de contraintes. Puisqu'on va être amené à changer Γ mais pas Dom, on notera $CSP(\Gamma)$ pour $CSP(\mathcal{B})$ dans la suite.

Exemple. Supposons que Γ contiennent deux relations binaires R_1 et R_2. Les deux contraintes $((x,z), R_1)$ et $((z,y), R_2)$ induisent une contrainte implicite $((x,y), R_3)$, où $R_3 = R_1 \circ R_2$. La relation R_3 n'est pas forcément dans Γ mais elle est *exprimable* par Γ. Il y a une réduction polynomiale de $CSP(\Gamma \cup \{R_3\})$ dans $CSP(\Gamma)$ puisqu'on peut remplacer chaque contrainte R_3 par un gadget composé de deux contraintes R_1 et R_2 et d'une variable additionnelle. Il existe une réduction triviale dans l'autre direction. Les deux problèmes $CSP(\Gamma)$ et $CSP(\Gamma \cup \{R_3\})$ sont donc polynomialement équivalents.

Il est important de noter que R_3 est l'interprétation de la formule $\exists z R_1(x,z) \wedge R_2(z,y)$.

On note par $\langle\Gamma\rangle$ l'ensemble des relations *exprimables* à partir de celles de Γ. Formellement, il s'agit de toutes les relations qu'on peut obtenir en interprétant des formules *primitives positives* sur Γ (il s'agit de formules contenant $\exists, \wedge, =$). Notez que $\langle\Gamma\rangle$ contient une infinité

7. $FPT \neq W[1]$: une conjecture analogue à $P \neq NP$ en complexité paramétrée [18].

de relations. Jusqu'ici nous ne considérions que des langages de contraintes finis. On dira qu'un langage infini est polynomial [8] ssi tous ses fragments finis le sont. Le théorème suivant nous permet de restreindre notre étude de la complexité aux langage de contraintes tels que $\Gamma = \langle\Gamma\rangle$.

Théorème 7 (Jeavons 1998 [25]). *Si Γ_1 et Γ_2 sont des langages de contraintes tels que $\langle\Gamma_1\rangle \subseteq \langle\Gamma_2\rangle$ alors $CSP(\Gamma_1)$ se réduit polynomialement [9] à $CSP(\Gamma_2)$.*

On peut considérer ces ensembles clos par $\langle\,.\,\rangle$, les *clones relationnels*, par rapport à l'inclusion et chercher parmi eux les langages Γ maximaux (respectivement minimaux) qui sont polynomiaux (respectivement NP-complets). Ceci revient à étudier la frontière entre cas faciles et difficiles dans un treillis connu sous le nom de *treillis des clones relationnels* en algèbre universelle. Dans le cas booléen, lorsque Dom a deux éléments, ce treillis est connu sous le nom de *treillis de Post* et les preuves modernes du théorème de Schaefer s'appuient très directement sur ce treillis. En fait le treillis de Post ne concerne pas des relations mais des fonctions. On passe de l'un à l'autre par la notion de *préservation* déjà entrevue et qu'on ne définira pas formellement ici.

Exemples. Si Γ est l'ensemble des relations correspondant à des clauses de Horn, alors toutes les relations de Γ sont préservées par l'opération booléenne $\wedge$. On dira dans ce cas que $\wedge$ est un *polymorphisme* de Γ.

Si Γ est l'ensemble des relations 0-valides, c'est-à-dire contenant un r-uplet avec seulement des 0, alors la fonction constante 0 est un polymorphisme de Γ.

On note par $\mathsf{Pol}(\Gamma)$ l'ensemble de tous les polymorphismes de Γ. Inversement, si F est un ensemble de fonctions [10], on note par $\mathsf{Inv}(F)$ l'ensemble des relations préservées (**inv**ariantes) par toutes les fonctions de F. Les opérateurs Pol et Inv établissent une correspondance de Galois entre deux treillis, l'un étant celui des clones relationnels, l'autre étant celui des *clones (fonctionnels)*. Puisque, plus il y a de relations à préserver, moins il y a de fonctions qui vont les préserver, et inversement, les deux treillis ont exactement la même structure, mais le haut de l'un correspond au bas de l'autre [11] (cf. Figure 2). En particulier, le langage de toutes les relations correspond au clone des fonctions triviales que sont les projections. Le clone

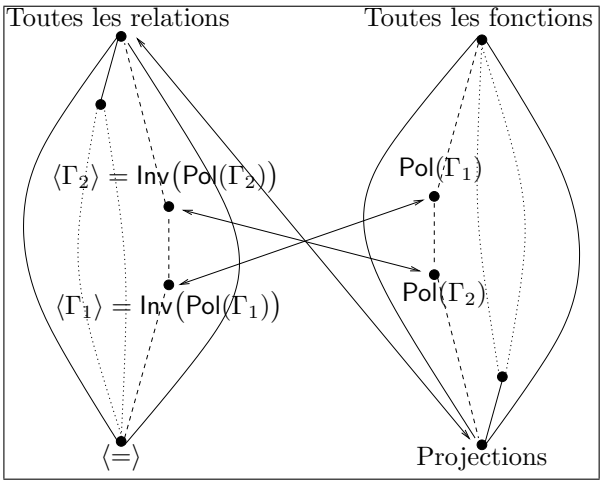


FIGURE 2 – Correspondance de Galois

de toutes les fonctions ne laisse *grosso modo* invariant que la relation $=$, et correspond au clone relationnel $\langle=\rangle$.

Intuitivement les polymorphismes contrôlent l'expressivité et donc la complexité. Comme les deux treillis sont structurellement les mêmes on va travailler sur le treillis des clones.

Les éléments du treillis de Post (cf. Figure 3) correspondent à des clones booléens, c'est-à-dire des ensemble de fonctions booléennes. Sur cette figure, on a tout en haut, toutes les fonctions booléennes (BF), tout en bas les projections (I_2). La classification de Schaefer (voir théorème 3) peut être vue comme un corollaire du théorème de Jeavons, de la description du treillis de Post et d'une étude de la complexité de 7 cas. Sur le treillis, ces 7 cas sont $I_0, I_1, V_2, E_2, L_2, D_2$ (qui correspondent aux 6 langages de contraintes polynomiaux et maximaux) et N_2 (qui est NP-complet et minimal).

Remarque. Ce qui est remarquable mais n'apparaît pas dans notre version aseptisée du théorème de Schaefer, c'est que chaque langage de contrainte maximalement polynomial est caractérisé en terme de préservation par une fonction booléenne. Ainsi, on peut disposer d'un algorithme général qui saura *identifier* les cas pour lesquels il existe un algorithme polynomial, et basculer s'il y a lieu en mode polynomial.

Lorsqu'on se penche sur les cas où le domaine a au moins trois valeurs, on peut de nouveau se ramener à l'étude du treillis des clones, mais contrairement à celui de Post, ces treillis sont grands (ils ne sont plus dénombrables), très complexes et largement inconnus par les algébristes. Jeavons et ses co-auteurs ont beaucoup travaillé à la généralisation des résultats de Schaefer. Ils démontrent par exemple précisément quand *établir la k-cohérence forte* permet de décider si

8. Pour être précis, il s'agit de *local tractability*. En pratique, des algorithmes vont « uniformiser » : par exemple, l'algorithme pour résoudre Horn-3-Sat est le même pour Horn-Sat. Ce second cas est dit *globally tractable*.

9. En fait, cette réduction est même dans logspace et préservera donc une analyse de complexité plus fine avec des classes comme L ou NL.

10. de Dom^r dans Dom, r étant une arité quelconque.

11. Pol et Inv sont des anti-isomorphismes de treillis.

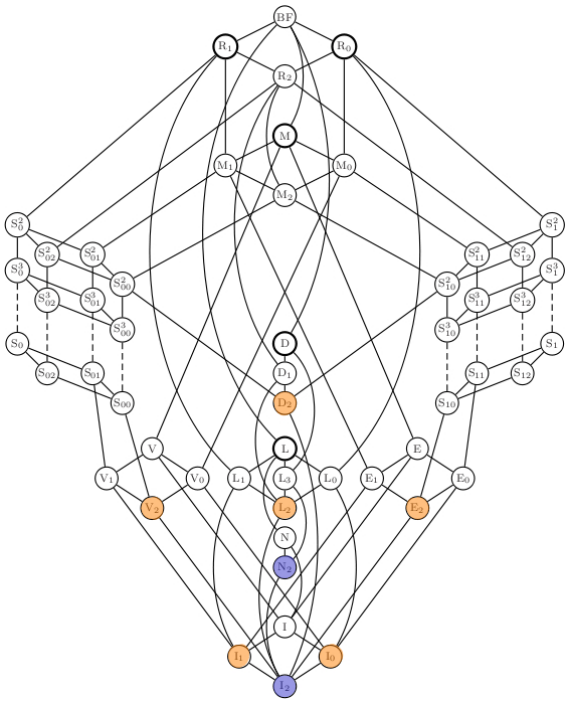


FIGURE 3 – Treillis de Post (1941).

une instance est satisfaisable [26] : à savoir, lorsque le langage des contraintes est préservé par une opération *near-unanimity* d'arité $k+1$.

Exemple. Le cas du langage 2-Sat déjà évoqué est un example pour $k = 2$: il y a préservation par la fonction majorité ternaire. La 2-cohérence (*path consistency*) est un algorithme complet pour le problème de décision dans ce cas restreint.

Feder et Vardi étudient ce même cas en utilisant la notion de programme Datalog, qui est intrinsèquement exécutable en temps polynomial [20]. Datalog est un cadre plus riche puisqu'il permet aussi d'expliquer le cas d'Horn-Sat de nature différente. Un programme Datalog calcule récursivement des relations auxiliaires. Intuitivement, c'est une forme de propagation de contraintes. On dérive localement de nouvelles contraintes à l'aide des règles du programme (l'optique étant de dériver une contradiction).

Exemple. Le programme suivant décide si une instance de Horn-3-Sat est insatisfaisable.

$$\begin{aligned} \lambda(x) &\leftarrow 1(x) \\ \lambda(x) &\leftarrow \lambda(y), \lambda(z), R(x,y,z) \\ \gamma &\leftarrow \lambda(x), 0(x) \end{aligned}$$

Le prédicat constant γ est le *but du programme* : si ce dernier « est activé » alors l'instance est rejetée.

Malgré ces avancés importantes, les résultats sont longtemps restés parcellaires et l'analyse sur le treillis ne permettait pas d'obtenir de classification complète. Il a fallu attendre Bulatov pour une classification complète du cas où il y a seulement *trois valeurs* [5]. Ce résultat peut sembler très incrémental mais conceptuellement il contient des idées très importantes qui permettent de s'abstraire du treillis et de travailler directement sur des variétés [12]. En effet, un langage de contrainte correspond à une algèbre et localement cette algèbre ne peut prendre que 5 types (par exemple elle se comporte comme un treillis ou encore comme une algèbre booléenne) : l'analyse de ces types permet d'établir la complexité associée au langage de contrainte. Ce renouveau de l'approche algébrique a permis à Bulatov d'établir une classification complète, *pour n'importe quel domaine fini*, de la complexité des langages de contraintes dits *conservatifs* [6]. Il s'agit du cas où on peut associer à chaque variable un domaine, une hypothèse qui semble être prévalente en intelligence artificielle.

Ce second résultat très important confirme ce que Feder et Vardi avaient suggéré dans leur papier fondateur [20], à savoir la pauvreté relative de « l'arsenal algorithmique polynomial ». Il existe essentiellement deux algorithmes, pour résoudre un problème non-uniforme : l'un généralise la cohérence et correspond à l'existence d'un programme Datalog, et l'autre découle du fait qu'on peut dans certain cas représenter de manière compacte l'ensemble des solutions [13].

Il existe une reformulation algébrique de la conjecture de la dichotomie et de nombreux raffinements pour d'autres classes de complexité. Depuis quelques années, l'approche algébrique des problèmes de contraintes a contribué à redynamiser l'algèbre universelle et à apporter de nouvelles questions dans cette discipline. Après une longue série de travaux, on sait par exemple maintenant que l'assertion suivante implique la conjecture de la dichotomie.

Conjecture 8. *Si* $\mathsf{Pol}(\Gamma)$ *contient un* terme de Siggers *alors* Γ *est polynomial.*

Un terme de Siggers est une opération d'arité 4 satisfaisant les identités $f(x,x,x,x) = x$ et $f(y,x,y,z) = f(x,y,z,x)$.

Pour plus de détails sur la dichotomie de la conjecture et la nouvelle approche algébrique, voir [9].

12. au sens de Birkhoff.
13. un exemple typique est la résolution d'équation linéaires.

Fragment	Dual	Classification ?
$\{\exists, \vee\}$ $\{\exists, \vee, =\}$	$\{\forall, \wedge\}$ $\{\forall, \wedge, \neq\}$	logspace
$\{\exists, \wedge, \vee\}$ $\{\exists, \wedge, \vee, =\}$	$\{\forall, \wedge, \vee\}$ $\{\forall, \wedge, \vee, \neq\}$	logspace si toutes les boucles, NP-complet sinon
$\{\exists, \wedge\}$ $\{\exists, \wedge, =\}$	$\{\forall, \vee\}$ $\{\forall, \vee, \neq\}$	**Conjecture de la dichotomie** des CSP (Ptime ou NP-complet). Résolue dans le cas des graphes (Hell et Nešetřil), le cas booléen (Schaefer), le cas à trois éléments (Bulatov) et le cas conservatif (Bulatov). Il suffit de trouver un algorithme polynomial pour les graphes orientés admettant un terme de Siggers pour résoudre la conjecture.
$\{\exists, \wedge, \neq\}$	$\{\forall, \vee, =\}$	NP-complet si $\|A\| \geq 3$, se réduit aux théorème de Schaefer sinon.
$\{\exists, \forall, \wedge\}$ $\{\exists, \forall, \wedge, =\}$	$\{\exists, \forall, \vee\}$ $\{\exists, \forall, \vee, \neq\}$	Une dichotomie entre Ptime et Pspace-complet est démontrée dans le cas booléen (Schaefer, Creignou). **En général, il n'y a pas de conjecture précise**. Des résultats partiels montrent que les complexités Ptime, NP-complet, et Pspace-complet sont atteintes. Le cas des graphes non orienté reste ouvert.
$\{\exists, \forall, \wedge, \neq\}$	$\{\exists, \forall, \vee, =\}$	Pspace-complet pour $\|A\| \geq 3$, se réduit à la classification de Schaefer pour QSat sinon.
$\{\forall, \exists, \wedge, \vee\}$		Pour QCSP avec disjonction, nous démontrons une tetrachotomie non triviale entre logspace, NP-complet, co-NP-complet et Pspace-complet [30].
$\{\forall, \exists, \wedge, \vee, =\}$ $\{\forall, \exists, \wedge, \vee, \neq\}$ $\{\neg, \exists, \forall, \wedge, \vee, =\}$		Ptime pour $\|A\| \leq 1$, Pspace-complet sinon.
$\{\neg, \exists, \forall, \wedge, \vee\}$		Ptime quand $\mathcal{B}$ ne contient que des relations vides ou complète , Pspace-complet sinon

FIGURE 4 – Complexité de CSP, QCSP et du model checking en général (pour lire la complexité du fragment dual, remplacer NP par co-NP et *vice versa*).

6 Autres questions

L'approche algébrique permet d'attaquer d'autres questions de complexité que la conjecture de la dichotomie. En particulier, dans le cas booléen, Nadia Creignou *et. al.* montrent qu'on peut s'appuyer sur le treillis de Post pour caractériser la complexité de nombreux problèmes [15]. Parfois, on dispose d'un théorème similaire au théorème de Jeavons et on peut travailler *a priori* sur le treillis, c'est la cas pour la *circumscription* ou l'*abduction*. Dans d'autres cas, on peut s'appuyer sur le treillis pour réaliser l'étude mais on ne remarque qu'*a posteriori* que le résultat est compatible avec la structure du treillis, par exemple pour le *dénombrement* ou *l'énumération* des solutions de Sat. D'autres questions importantes ne sont pas compatibles avec la structure du treillis, c'est le cas par exemple de MaxSat, de la complexité paramétrée de Sat ou encore de son approximation.

Pour des problèmes qui généralisent CSP, par exemple les *problèmes de contraintes quantifiées* (QCSP), on peut adapter la méthode algébrique et travailler avec des clones relationnels particuliers, correspondant à une clôture par interprétation avec les symboles $\exists, \wedge, =$ mais aussi avec $\forall$. Puisque les clones relationnels ont plus de relations, ceci signifie qu'on aura moins de polymorphismes. Dans ce cas, les *polymorphismes surjectifs* caractérisent la complexité [3]. Pour le cas des QCSP booléens, la classification est complète, toujours à l'aide du treillis de Post [13]. En général, La classification de QCSP reste incomplète et est au moins aussi difficile que la conjecture de la dichotomie. On observe jusqu'à présent une *trichotomie* avec les complexités P, NP-complet et Pspace-complet.

On reformule souvent QCSP comme le problème de *model checking* pour le fragment de la logique du premier ordre utilisant les symboles $\exists, \forall, \wedge, =$. Le *model checking problem* prend en entrée une formule ϕ et en paramètre une structure $\mathcal{B}$ et demande si $\mathcal{B}$ est un modèle de ϕ. Comme pour le CSP non-uniforme, on considère que $\mathcal{B}$ est fixé et on cherche à connaître la complexité du problème pour chaque $\mathcal{B}$. Martin a étudié la complexité de ce problème pour de nombreux fragments de la logique du premier ordre [33]. Il a établi que soit un fragment est facilement classifiable, ou se réduit au théorème de Schaefer, soit il s'agit de fragments riches du point de vue de la complexité (voir Figure 4). Il reste 3 cas : CSP $(\exists, \wedge, =)$, QCSP $(\forall, \exists, \wedge, =)$ et *QCSP avec disjonction* $(\forall, \exists, \wedge, \vee)$. Dans le cas de CSP ou QCSP la présence ou l'absence du symbole $=$ n'a pas d'effet sur la complexité. Par contre dans ce dernier cas, l'absence d'égalité est cruciale (en présence de l'égalité le problème est facilement classifiable, il est

Pspace-complet si $\mathcal{B}$ a deux éléments ou plus et dans P sinon).

D'autres correspondances de Galois que la correspondance $\mathsf{Pol} - \mathsf{Inv}$ ont étés étudiées en algèbre universelle. Il n'y a pas vraiment de méthode générique pour démontrer les théorèmes techniques nécessaires mais Ferdinand Börner propose des recettes assez complètes de mise en œuvre [2]. Il s'avère que dans le cas de *QCSP avec disjonction* $(\forall, \exists, \wedge, \vee)$ puisqu'on ne dispose plus de l'égalité, on ne travaille plus avec une notion de préservation par une fonction comme pour les polymorphismes avec CSP mais avec des *hyper-fonctions* : c'est-à-dire des fonctions de Dom dans l'ensemble de ses parties, ce qui rend les choses plus complexes. D'un autre côté, la présence de $\vee$ signifie qu'on peut se contenter des hyper-fonctions qui sont unaires ; et, celle de $\forall$, comme pour le cas de QCSP, signifie qu'on peut imposer une notion adéquate de *surjectivité*.

L'expressivité de $\mathcal{B}$ et donc sa complexité pour le problème de QCSP avec disjonction est caractérisée par ses *hyper-endomorphismes surjectifs*. Pour chaque valeur finie de Dom, on se ramène à l'étude d'un treillis *fini*. Pour le cas booléen, celui-ci est trivial et on observe une dichotomie entre Pspace-complet et Logspace. Pour le cas à trois éléments, on peut calculer les éléments importants du treillis *à la main* et on obtient une tetrachotomie entre Logspace, NP-complete, Co-NP-complete et Pspace-complete. De plus, la complexité s'explique de manière uniforme par la possibilité d'*éliminer les quantificateurs* (par exemple, élimination des $\forall$ mais pas des $\exists$ donne un problème NP-complet) [31]. Pour quatre éléments, l'explosion combinatoire empêche de faire ce calcul à la main et en passant à une preuve assistée par ordinateur, on peut démontrer une tetrachotomie [34].

Finalement, en définissant une notion adéquate de *cœur quantifié*, en terme de propriétés de relativisation, et en le caractérisant algébriquement, on est capable de faire abstraction du treillis et de se ramener à l'étude de cas génériques semblables au cas à 4 éléments. On obtient ainsi une tetrachotomie pour n'importe quel domaine [30]. Le méta-problème est NP-complet, ce qui n'est pas forcément insurmontable pour un problème dont la complexité est Pspace-complet.

7 Conclusion

Classifier la complexité des problèmes de satisfaction de contraintes est une question centrale en informatique théorique qui a des liens forts avec les mathématiques (algèbre, combinatoire, logique) et qui en informatique dépasse largement le cadre de l'intelligence artificielle du fait des liens importants avec les bases de données. Mathématiquement très riche et très actif, ce domaine dépasse le cadre de la question de la dichotomie, même si cette conjecture est importante.

Nous avons fait un rapide survol de différents résultats théoriques. Nous avons vu que deux types de restrictions permettant d'obtenir un problème polynomial pour une instance $(\mathcal{A}, \mathcal{B})$: soit le graphe des contraintes $\mathcal{A}$ est arborescent, soit le langage des contraintes $\mathcal{B}$ présente de bonnes propriétés algébriques. Plus récemment, la question des *classes polynomiales hybrides* a pris de l'importance : il s'agit d'identifier des classes polynomiales nouvelles où $\mathcal{A}$ et $\mathcal{B}$ sont restreints simultanément (voir par exemple [11]).

Même si du point de vue pratique on pourrait reprocher à certaines hypothèses d'être irréalistes, de nombreux concepts ont un impact concret. Par exemple, la notion de cœur d'une structure et la notion de paire duale ont des applications en bases de données dans le cadre de l'intégration d'information [37, 38]. Par ailleurs, on voit apparaître des résultats qui font des hypothèses plus réalistes, par exemple sur la manière dont les contraintes sont codées [10], ou qui prennent en compte des contraintes globales [8].

Références

[1] M. Bodirsky. Constraint satisfaction problems with infinite templates. In Creignou et al. [14], pages 196–228.

[2] F. Börner. Basics of galois connections. In Creignou et al. [14], pages 38–67.

[3] F. Börner, A. A. Bulatov, H. Chen, P. Jeavons, and A. A. Krokhin. The complexity of constraint satisfaction games and QCSP. *Inf. Comput.*, 207(9) :923–944, 2009.

[4] A. A. Bulatov. H-coloring dichotomy revisited. *Theor. Comput. Sci.*, 349(1) :31–39, 2005.

[5] A. A. Bulatov. A dichotomy theorem for constraint satisfaction problems on a 3-element set. *J. ACM*, 53(1) :66–120, 2006.

[6] A. A. Bulatov. Complexity of conservative constraint satisfaction problems. *ACM Trans. Comput. Log.*, to appear.

[7] A. A. Bulatov, A. A. Krokhin, and B. Larose. Dualities for constraint satisfaction problems. In Creignou et al. [14], pages 93–124.

[8] A. A. Bulatov and D. Marx. Constraint satisfaction problems and global cardinality constraints. *Commun. ACM*, 53(9) :99–106, 2010.

[9] A. A. Bulatov and M. Valeriote. Recent results on the algebraic approach to the csp. In Creignou et al. [14], pages 68–92.

[10] H. Chen and M. Grohe. Constraint satisfaction with succinctly specified relations. *J. Comput. Syst. Sci.*, 76(8) :847–860, 2010.

[11] M. C. Cooper, P. G. Jeavons, and A. Z. Salamon. Generalizing constraint satisfaction on trees : Hybrid tractability and variable elimination. *Artif. Intell.*, 174(9-10) :570–584, 2010.

[12] B. Courcelle. Graph rewriting : An algebraic and logic approach. In *Handbook of Theoretical Computer Science, Volume B : Formal Models and Sematics (B)*, pages 193–242. 1990.

[13] N. Creignou, S. Khanna, and M. Sudan. *Complexity classifications of boolean constraint satisfaction problems*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2001.

[14] N. Creignou, P. G. Kolaitis, and H. Vollmer, editors. *Complexity of Constraints - An Overview of Current Research Themes [Result of a Dagstuhl Seminar]*, volume 5250 of *Lncs*. Springer, 2008.

[15] N. Creignou and H. Vollmer. Boolean constraint satisfaction problems : When does post's lattice help ? In Creignou et al. [14], pages 3–37.

[16] V. Dalmau, P. G. Kolaitis, and M. Y. Vardi. Constraint satisfaction, bounded treewidth, and finite-variable logics. In P. Van Hentenryck, editor, *CP*, volume 2470 of *Lncs*, pages 310–326. Springer, 2002.

[17] R. Dechter. Bucket elimination : A unifying framework for reasoning. *Artif. Intell.*, 113(1-2) :41–85, 1999.

[18] R. G. Downey and M.R. Fellows. *Parameterized Complexity*. Monographs in Computer Science. Springer, 1999.

[19] T. Feder, F. R. Madelaine, and I. A. Stewart. Dichotomies for classes of homomorphism problems involving unary functions. *Theor. Comput. Sci.*, 314(1-2) :1–43, 2004.

[20] T. Feder and M. Y. Vardi. The computational structure of monotone monadic SNP and constraint satisfaction : a study through datalog and group theory. *SIAM J. Comput.*, 28, 1999.

[21] E. C. Freuder. A sufficient condition for backtrack-bounded search. *J. ACM*, 32(4) :755–761, 1985.

[22] M. Grohe. The complexity of homomorphism and constraint satisfaction problems seen from the other side. *J. ACM*, 54(1), 2007.

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

[24] P. Hell and J. Nešetřil. *Graphs and Homomorphisms*. OUP, 2004.

[25] P. Jeavons. On the algebraic structure of combinatorial problems. *Theoretical Computer Science*, 200(1–2) :185–204, 1998.

[26] P. Jeavons, D. Cohen, and M. Gyssens. Closure properties of constraints. *J. ACM*, 44(4) :527–548, 1997.

[27] R. E. Ladner. On the structure of polynomial time reducibility. *J. ACM*, 22(1) :155–171, 1975.

[28] F. R. Madelaine. Universal structures and the logic of forbidden patterns. *Logical Methods In Computer Science*, 5(2), 2009.

[29] F. R. Madelaine. On the containment of forbidden patterns problems. In *CP*, pages 345–359, 2010.

[30] F. R. Madelaine and B. Martin. A tetrachotomy for positive first-order logic without equality. In *Logic in Computer Science*. IEEE Computer Society, 2011, à paraître.

[31] F. R. Madelaine and B. Martin. The complexity of positive first-order logic without equality. *ACM Trans. Comput. Log.*, to appear.

[32] F. R. Madelaine and I. A. Stewart. Constraint satisfaction, logic and forbidden patterns. *SIAM J. Comput.*, 37(1) :132–163, 2007.

[33] B. Martin. First-order model checking problems parameterized by the model. In *CiE*, pages 417–427, 2008.

[34] B. Martin and J. Martin. The complexity of positive first-order logic without equality II : The four-element case. In *CSL*, pages 426–438, 2010.

[35] D. Marx. Tractable hypergraph properties for constraint satisfaction and conjunctive queries. In L. J. Schulman, editor, *STOC*, pages 735–744. ACM, 2010.

[36] F. Scarcello, G. Gottlob, and G. Greco. Uniform constraint satisfaction problems and database theory. In Creignou et al. [14], pages 156–195.

[37] B. ten Cate, L. Chiticariu, P. G. Kolaitis, and W. Chiew Tan. Laconic schema mappings : Computing the core with sql queries. *PVLDB*, 2(1) :1006–1017, 2009.

[38] B. ten Cate, P. G. Kolaitis, and W. Chiew Tan. Database constraints and homomorphism dualities. In D. Cohen, editor, *CP*, volume 6308 of *Lncs*, pages 475–490. Springer, 2010.

Programmation par contraintes pour l'optimisation de plans de test de satellites

Caroline Maillet Gérard Verfaillie Bertrand Cabon

ONERA
2 avenue Edouard Belin, BP 4025, 31055 Toulouse Cedex, France
Astrium
31 rue des Cosmonautes, Z.I. des Palays, 31402 Toulouse Cedex 4, France
{Caroline.Maillet,Gerard.Verfaillie}@onera.fr, {Caroline.Maillet,Bertrand.Cabon}@astrium.eads.net

Résumé

Les tests physiques des charges utiles des satellites de télécommunication en chambre à vide durent plusieurs semaines et mobilisent des équipes importantes. Ces tests sont organisés selon des transitions thermiques permettant chacune de tester un sous ensemble d'équipements sous différentes contraintes. L'objectif de réduction du nombre de transitions thermiques est un enjeu crucial pour Astrium. Pour modéliser ce problème, nous avons élaboré un modèle en programmation par contraintes. Ce modèle peut induire des milliers de variables sur lesquelles s'appliquent plusieurs centaines de milliers de contraintes binaires et n-aires. Pour le résoudre, nous avons développé des algorithmes spécifiques utilisant des mécanismes de retour-arrière dirigé par les conflits, des heuristiques adaptatives originales pondérant les variables et des mécanismes de redémarrage. Les expérimentations ont montré que ces mécanismes apportent des gains significatifs en termes de qualité des plans produits en un temps de calcul borné.

Abstract

Communication satellite payload tests last several weeks in thermal vacuum chambers, and mobilise large validation teams. These tests are organised over several thermal transitions, with each transition allowing the testing of a subset of the equipment under different constraints. Minimising the number of transitions is a crucial objective for Astrium. Constraint Programming was used to elaborate a mathematical model of the problem. This model may induce thousands of variables with hundreds of thousands of binary and n-ary constraints. To solve it, specific algorithms were developed, using Conflict Directed Backjumping mechanisms, new adaptive variable weighting heuristics, and restarting strategies. Experiments showed significative gains in terms of quality of the plans produced within a limited computing time.

1 Contexte industriel

Astrium est la filiale d'EADS en charge de la conception et de la fabrication de satellites d'observation et de télécommunication. Les satellites de télécommunication sont des relais géostationnaires qui assurent des services tels que la téléphonie, la télévision haute définition ou encore internet. Situé à 36 000 km de la Terre, ils reçoivent avec une très faible puissance les signaux radio fréquence (RF) émis par les stations sol. Les signaux reçus sont amplifiés, convertis en fréquence pour éviter les interférences entre signaux reçus et signaux émis. Ils sont enfin filtrés et réamplifiés avant d'être émis vers le sol. Pour assurer cette mission, les charges utiles des satellites de télécommunication sont constituées d'un grand nombre d'amplificateurs, de filtres, de convertisseurs de fréquence et de connections reliant ces différents équipements. Tous ces équipements sont redondés pour permettre des reconfigurations en cas de pannes. La charge utile d'un satellite de télécommunication dépend de la mission à assurer. L'augmentation de la complexité des missions (nombre de signaux RF) se traduit par une augmentation de la complexité des charges utiles.

Avant lancement, les milliers d'équipements d'une charge utile doivent être testés dans des conditions proches de l'environnement spatial. Pour cela, l'essentiel des tests sont réalisés en chambre à vide dans des conditions extrêmes en température. Les tests réalisés sont organisés suivant une séquence de transitions

thermiques ascendantes ou descendantes en température. A chacune des transitions thermiques correspond une certaine configuration de la charge utile. Chaque transition permet de satisfaire une partie des exigences de test définies par Astrium. L'ensemble des transitions doit assurer la couverture de l'ensemble des exigences de test.

Les opérations de test associées à une transition durent plus d'une dizaine d'heures. Au total, ces opérations de test mobilisent sur plusieurs semaines des équipes importantes présentes 24h/24. L'objectif de limitation du nombre de transitions thermiques permettant d'assurer la couverture de l'ensemble des exigences de test est donc un objectif crucial pour Astrium.

2 Problème d'optimisation des plans de test des charges utiles de télécommunication

A chaque transition thermique correspond une configuration de la charge utile. A une configuration est associé un ensemble des chemins. Un chemin correspond à un signal RF traversant la charge utile depuis une antenne réceptrice vers une antenne émettrice et passant par un ensemble d'équipements (amplificateurs, filtres, convertisseurs, routeurs et connexions entre eux). Les routeurs sont des équipements configurables qui permettent de diriger les signaux RF vers les équipements appropriés. Par exemple, la figure 1 montre une partie du chemin rouge qui utilise les routeurs 3508 et 4227, l'amplificateur 5005, les routeurs 7988 et 7990 et le filtre CH17. Lors d'une transition thermique, tous les chemins de la configuration associée et donc tous les équipements présents sur ces chemins peuvent être testés en parallèle.

Cependant l'ensemble des chemins associés à une transition doit respecter certaines contraintes.

Premièrement, tous les chemins doivent être compatibles deux à deux en termes de positions des routeurs. Par exemple sur la figure 1, les chemins rouge et bleu sont compatibles car ils n'utilisent aucun équipement en commun. Les chemins vert et bleu sont aussi compatibles car le seul équipement commun est le routeur 3507 utilisé dans la même position par les deux chemins. Par contre, les chemins rouge et vert sont incompatibles car ils utilisent tous deux le routeur 7990 dans des positions différentes.

Deuxièmement, du fait de contraintes thermiques et de contraintes de mesures, chaque transition est limitée en capacité : le nombre de chemins associés doit être inférieur à un nombre maximum.

Troisièmement, du fait de contraintes thermiques locales, chaque zone de la charge utile est soumise à des limitations thermiques : le nombre d'équipements allumés doit rester inférieur à un nombre maximum. Par exemple, seuls deux équipements parmi les amplificateurs 5005, 5006, 5007, 5008 et 5009 de la figure 1, peuvent être utilisés par des chemins sur une transition descendante en température.

Enfin, certains chemins doivent être obligatoirement testés et donc apparaître dans au moins une transition. Pour d'autres chemins, certaines transitions sont interdites.

Les exigences de test portent sur l'ensemble des transitions thermiques. Elles prennent différentes formes.

Les plus courantes imposent de tester un équipement dans au moins une transition thermique, ce qui impose qu'au moins un chemin passant par cet équipement soit testé dans au moins une transition thermique. Par exemple sur la figure 1, l'utilisation du chemin rouge ou du chemin vert permet de satisfaire l'exigence de test du routeur 7990. Au minimum, l'ensemble des équipements de la charge utile doit être testé dans au moins une transition thermique.

D'autres exigences imposent de réaliser un nombre minimum (supérieur à un) de tests de certains équipements.

D'autres imposent de tester un équipement sous certaines conditions thermiques, ce qui limite les transitions utilisables.

D'autres enfin imposent de tester un équipement sous certaines conditions fréquentielles, ce qui limite les chemins utilisables. Par exemple, sur la figure 1, si le routeur 7990 doit être testé en fréquence haute, le chemin vert qui passe par le filtre CH11 de fréquence trop basse ne permet pas de satisfaire cette exigence. Par contre, le chemin rouge qui passe par le filtre CH17 de fréquence haute permet de la satisfaire.

A noter que l'appartenance d'un chemin à une transition permet de satisfaire un ensemble d'exigences portant sur les équipements utilisés par ce chemin.

Le problème d'optimisation des plans de test peut être envisagé selon deux objectifs différents :

- la minimisation du nombre de transitions thermiques utilisées sous contrainte de satisfaction de l'ensemble des exigences de test,
- la maximisation du nombre d'exigences de test satisfaites à nombre de transitions thermiques utilisées fixé.

Comme le nombre de transitions nécessaires est de l'ordre de la dizaine, il est possible d'itérer manuellement sur ce nombre. C'est pourquoi l'étude s'est

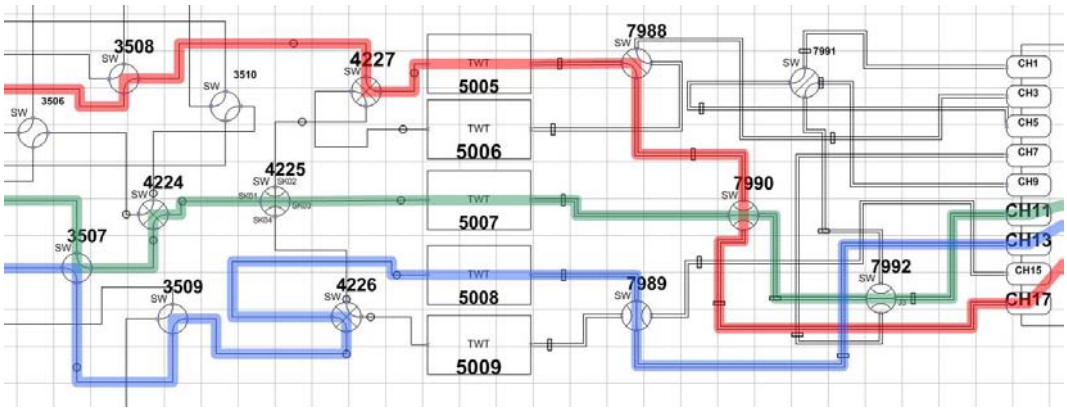


FIGURE 1 – Extrait d'une charge utile de satellite de télécommunication

focalisée sur le second objectif : maximisation du nombre d'exigences de test satisfaites à nombre de transitions fixé.

Les données de ce problème sont les suivantes :
- un nombre de transitions utilisables (de l'ordre de la dizaine),
- un ensemble de chemins utilisables (plusieurs milliers),
- un ensemble d'exigences de test (plusieurs milliers),
- un ensemble de contraintes sur les chemins utilisés :
  - contraintes binaires de compatibilité entre chemins (plusieurs centaines de milliers),
  - contraintes n-aires de capacité et de limitation thermique sur les transitions (plusieurs centaines),
  - contraintes unaires de restriction sur l'affectation des chemins aux transitions (plusieurs centaines).

La sortie du problème est un plan de test qui indique pour chaque chemin la transition dans laquelle il apparaît, éventuellement nulle s'il n'apparaît dans aucune transition. Cette sortie doit être produite en quelques minutes seulement pour répondre au besoin des utilisateurs.

A notre connaissance, ce type d'application n'a pas été traité dans la littérature scientifique. Le problème s'apparente à un problème de sacs à dos multiples et multidimensionnels avec conflits [14, 6]. Les transitions thermiques sont les sacs à dos et les chemins sont les objets. Les conflits sont les contraintes binaires d'incompatibilités entre chemins et les contraintes unaires de restrictions sur l'affectation des chemins. Les dimensions sont les contraintes n-aires de capacité et de limitation thermique des transitions. Par contre classiquement dans les problèmes de sac à dos, les utilités des objets sont supposées être additives. Ce n'est pas le cas dans notre problème : l'utilité d'un chemin est définie par l'ensemble des exigences de test qu'il peut satisfaire. Une exigence de test peut être satisfaite de façon équivalente par plusieurs chemins. Ces utilités ne sont pas additives. C'est pourquoi nous avons créé un modèle mathématique dédié au problème d'optimisation des plans de test des charges utiles de télécommunication.

3 Modèle mathématique

Dans un premier temps, nous avons construit un modèle mathématique en Programmation Par Contraintes (PPC) [5] utilisant CP Optimizer [1]. Ce modèle associe une variable à chaque chemin qui représente la transition dans lequel ce chemin est placé. Son domaine de valeurs est l'ensemble des transitions utilisables auquel on ajoute la valeur 0 pour représenter l'absence de sélection du chemin.

3.1 Données

- Nombre de chemins : P
- Nombre d'équipements : U
- Appartenance des équipements aux chemins :

$$UP[u][p], \quad \forall u \in \{1, \ldots, U\}, \forall p \in \{1, \ldots, P\}$$

- Nombre de transitions thermiques : T
- Capacité des transitions en nombre de chemins :

$$CAP[t], \quad \forall t \in \{1, \ldots, T\}$$

- Compatibilité entre chemins :

$$COMP[p][p'], \quad \forall p, p' \in \{1, \ldots, P\}$$

– Restriction d'affectation des chemins aux transitions :

$$REST[p][t], \quad \forall p \in \{1, \ldots, P\}, \forall t \in \{0, \ldots, T\}$$

– Nombre d'exigences de test : R
– Nombre minimum de chemins à sélectionner pour satisfaire les exigences de test :

$$MIN\_R[r], \quad \forall r \in \{1, \ldots, R\}$$

– Appartenance des chemins aux exigences de test :

$$PR[p][r], \quad \forall p \in \{1, \ldots, P\}, \forall r \in \{1, \ldots, R\}$$

– Appartenance des transitions aux exigences de test :

$$TR[t][r], \quad \forall t \in \{1, \ldots, T\}, \forall r \in \{1, \ldots, R\}$$

– Nombre de limitations thermiques : L
– Nombre maximum d'équipements utilisés par transition pour respecter les limitations thermiques :

$$MAX\_L[l], \quad \forall l \in \{1, \ldots, L\}$$

– Appartenance des équipements aux limitations thermiques :

$$UL[u][l], \quad \forall u \in \{1, \ldots, U\}, \forall l \in \{1, \ldots, L\}$$

– Appartenance des transitions aux limitations thermiques :

$$TL[t][l], \quad \forall t \in \{1, \ldots, T\}, \forall l \in \{1, \ldots, L\}$$

3.2 Variables

Pour chaque chemin, la transition thermique utilisée ou la valeur 0 pour désigner l'absence de sélection :

$$aff[p] \in \{0, \ldots, T\}, \quad \forall p \in \{1, \ldots, P\}$$

3.3 Expressions

– Pour chaque équipement, le fait qu'il soit utilisé ou pas :

$$\forall u \in \{1, \ldots, U\}, \forall t \in \{1, \ldots, T\},$$
$$usedU[u][t] = \Big(\sum_{p=1}^{P/UP[u][p]=1} (aff[p] = t) \Big) > 0$$

– Pour chaque exigence de test, le nombre de chemins sélectionnés satisfaisant cette exigence :

$$\forall r \in \{1, \ldots, R\},$$
$$testR[r] = \sum_{p=1,t=1}^{\substack{P/PR[p][r]=1,\\ T/TR[t][r]=1}} (aff[p] = t)$$

– Pour chaque exigence de test, son niveau de satisfaction :

$$\forall r \in \{1, \ldots, R\},$$
$$satR[r] = \min(testR[r], MIN\_R[r])$$

– Critère à optimiser (somme des niveaux de satisfaction des exigences de test) :

$$obj = \sum_{r=1}^{R} satR[r]$$

3.4 Objectif

maximize obj

3.5 Contraintes

Contraintes unaires Restriction d'affectation des chemins aux transitions :

$$\forall p \in \{1, \ldots, P\}, \forall t \in \{0, \ldots, T\} / REST[p][t] = 0, \quad aff[p] \neq t$$

Contraintes binaires Compatibilité entre chemins affectés à une même transition :

$$\forall p, \forall p' \in \{1, \ldots, P\} / C[p][p'] = 0,$$
$$\Big(aff[p] \neq aff[p']\Big) \| \Big((aff[p] = 0) \&\& (aff[p'] = 0)\Big)$$

Contraintes n-aires

– Capacité de chaque transition :

$$\forall t \in \{1, \ldots, T\}, \quad \sum_{p=1}^{P} (aff[p] = t) \leq CAP[t]$$

– Limitations thermiques :

$$\forall l \in \{1, \ldots, L\}, \forall t \in \{1, \ldots, T\} / TL[t][l] = 1,$$
$$\sum_{u=1}^{U/UL[u][l]=1} (usedU[u][t] = 1) \leq MAX\_L[r]$$

A l'aide de CP Optimizer [1], ce modèle mathématique a été expérimenté sur des instances correspondant aux satellites en cours de conception. Sur des instances de taille modeste, il permet de produire rapidement des solutions optimales et parfois d'en prouver l'optimalité. Un autre modèle en Programmation Linéaire en Nombre Entiers [11] impliquant des variables binaires a également été expérimenté avec CPLEX [1]. Sur les mêmes instances, il permet plus souvent de prouver l'optimalité au prix d'un temps de résolution nettement plus élevé. Sur des instances de plus

grande taille, ces solveurs génériques (CP Optimizer et CPLEX) se heurtent à des problèmes d'insuffisance de mémoire. Les mêmes problèmes auraient été rencontrés avec des solveurs SAT (nombre astronomique de clauses induites par les contraintes n-aires de capacité et de limitation thermique, fonction exponentielle du nombre de chemins). C'est pourquoi nous avons fait le choix de développer un algorithme spécifique. Pour éviter les problèmes de mémoire, cet algorithme gère les contraintes de façon implicite grâce à des opérations sur des vecteurs de bits. Il réutilise par ailleurs de nombreuses techniques de PPC.

4 Algorithmes

L'algorithme spécifiquement développé par Astrium pour résoudre ce problème d'optimisation est un algorithme de recherche arborescente en profondeur d'abord, comportant :

- un mécanisme de propagation à base de Forward Checking [12],
- un calcul de majorant mettant à jour le domaine de la variable objectif *obj* (voir §3.3),
- un mécanisme de retour-arrière à base de Backtrack chronologique (BT) [3],
- un ordonnancement dynamique des variables,
- un ordonnancement dynamique des valeurs suivant une valeur décroissante de l'impact immédiat sur la valeur minimum de la variable *obj*.

Ces mécanismes ont été ensuite améliorés en ayant recours à :

- des mécanismes de retour-arrière intelligent : Backjumping (BJ) [7], Conflict Directed Backjumping (CBJ) [18, 21] avec production d'explications sur contraintes binaires et n-aires,
- des heuristiques d'ordonnancement de variables dynamiques, spécifiques et adaptatives : apprentissage des impacts des contraintes sur les variables inspiré du weighted degree (wdeg) [4] et du wvar [13], utilisation du dernier conflit [16],
- un mécanisme de redémarrage (Rs) : Restart avec bruitage des heuristiques [9, 22], Restart adaptatif avec utilisation des impacts [10] et Restart adaptatif bruité.

Bien que le mécanisme de propagation à base de Maintien d'Arc Cohérence (MAC) [20] soit généralement très performant, il n'a pas été implémenté dans ce solveur spécifique. A cause de la présence de la valeur 0 dans le domaine de la plupart des variables, toute valeur a un support suivant toute contrainte et le problème initial est arc-cohérent. Pour les mêmes raisons, pratiquement à chaque noeud de l'arbre de recherche le problème est arc-cohérent après filtrage par Forward Checking.

4.1 Mécanismes de retour-arrière

Trois mécanismes de retour-arrière ont été implémentés. Pour chacun de ces mécanismes, on appelle conflit le fait que l'affectation d'une valeur à une variable soit incohérente avec l'affectation courante. Un conflit sur l'affectation a d'une valeur à une variable est détecté soit par la propagation d'une affectation précédente, soit par la propagation de a qui a vidé le domaine d'une autre variable, soit par le fait que a produise un sous-arbre en échec. On appelle échec le fait que toutes les valeurs d'une variable soient en conflit. Quelle que soit la raison de l'échec, il faut remonter dans l'arbre pour débloquer le problème.

Le Backtrack (BT) [3] consiste à revenir sur l'affectation de la variable précédente.

Le Backjumping (BJ), introduit par J. Gaschnig [7], consiste à revenir sur la variable la plus récente impliquée dans le dernier conflit ayant conduit à l'échec. En cas d'échecs imbriqués, le premier retour-arrière est un backjump, les autres sont des backtracks classiques.

Le Conflict Directed Backjumping (CBJ), introduit par P. Prosser [18] et par T. Schiex et G. Verfaillie [21], mémorise l'explication de chaque conflit. Cette explication est constituée de l'ensemble des variables affectées responsables du conflit. Lors de l'échec d'une variable, un backjump est effectué sur la variable la plus récente appartenant aux explications de la variable en échec. Le CBJ permet de multiples backjumps imbriqués.

4.2 Heuristiques d'ordonnancement des variables

4.2.1 Heuristiques d'ordonnancement dynamiques et spécifiques

En PPC, les heuristiques dynamiques d'ordonnancement de variables permettent de choisir la future variable à affecter en fonction des affectations précédentes. Ces heuristiques favorisent les variables menant le plus probablement à un échec. Une heuristique classique reposant sur ce principe d'échec d'abord est $MinDomain$ [12]. Elle choisit dynamiquement la variable ayant la plus petite taille de domaine courant. Cependant, l'existence de la valeur 0 dans le domaine des variables rend cette heuristique peu pertinente car il faut donner la priorité aux variables n'ayant pas 0 dans leur domaine (sélection obligatoire dans une des transitions) par rapport aux variables ayant la valeur 0 (possibilité de non sélection).

Pour les variables n'ayant pas la valeur 0 dans leur domaine, l'heuristique $MinDomain$ est pertinente. Pour les autres, on peut chercher à s'inspirer des heuristiques utilisées dans les problèmes de sac à dos. L'heuristique la plus classique consiste à ordonnan-

cer statiquement les objets suivant une valeur décroissante du ratio entre leur utilité et leur poids [14]. Dans notre problème, il est difficile de définir ce que sont l'utilité et le poids d'un chemin. Pour cette raison, les variables ayant la valeur 0 dans leur domaine sont ordonnancées dynamiquement suivant une valeur décroissante de l'impact immédiat sur le critère : heuristique $MaxObjectif$. Cet impact est la valeur maximale de l'augmentation du niveau de satisfaction des exigences qu'il est possible d'obtenir en affectant une valeur à cette variable.

En s'inspirant de ces principes, une heuristique dynamique et spécifique d'ordonnancement des variables a été implémentée. Trois types de variables sont identifiées et ordonnancées de manière hiérarchique :

1. les variables ayant une seule valeur dans leur domaine sont affectées d'abord,
2. les variables n'ayant pas la valeur 0 dans leur domaine suivant l'heuristique $MinDomain$,
3. les autres variables ayant la valeur 0 dans leur domaine suivant l'heuristique $MaxObjectif$.

Cet ordonnancement a été raffiné afin de prendre en compte la compatibilité initiale des chemins : pour chaque chemin le nombre de chemins avec lesquels il est compatible. Cette information permet de choisir la variable la moins (resp. la plus) compatible avec les autres variables : heuristique $MinCompatible$ (resp. $MaxCompatible$). L'heuristique dynamique et spécifique (HDS) résultante est :

1. les variables ayant une seule valeur dans leur domaine sont affectées d'abord,
2. les variables n'ayant pas la valeur 0 dans leur domaine suivant $MinDomain$,
3. les variables n'ayant pas la valeur 0 dans leur domaine et ayant la même taille de domaine suivant $MinCompatible$,
4. les variables ayant la valeur 0 dans leur domaine suivant $MaxObjectif$,
5. les variables ayant la valeur 0 dans leur domaine et ayant le même impact sur le critère suivant $MaxCompatible$.

4.2.2 Heuristiques d'ordonnancement dynamiques, spécifiques et adaptatives

HDS ne prend pas en compte les contraintes n-aires du problème. Ces contraintes sont gérées de façon implicite pour éviter les problèmes de dépassement de mémoire. De ce fait, il n'est pas envisageable d'utiliser des heuristiques basées sur le degré ou le degré dynamique des variables [2], qui reposent sur des calculs de nombre de contraintes pesant sur les variables et qui favorisent les variables impliquées dans le plus grand nombre de contraintes.

Heuristique basée sur la pondération des variables responsables des échecs Dans un premier temps, nous avons recherché une heuristique permettant de prendre en compte les impacts des contraintes (binaires ou n-aires), telle que le weighted degree (wdeg) [4] et le wvar [13]. Le wdeg et le wvar ont été définis dans le cadre du Forward Checking avec Backtrack. Pour le wdeg, un poids est associé à chaque contrainte. Ce poids est incrémenté chaque fois que la propagation d'une contrainte provoque un échec. Le poids d'une variable est alors défini comme la somme des poids des contraintes actives pesant sur cette variable (contraintes ayant au moins deux variables non affectées). Pour le wvar, un poids est associé à chaque variable. Ce poids est incrémenté chaque fois que la variable est en échec. Ces apprentissages sont utilisés lors de l'ordonnancement dynamique des variables pour pondérer la taille du domaine : les heuristiques $Min\frac{Domain}{wdeg}$ ou $Min\frac{Domain}{wvar}$ remplacent l'heuristique $MinDomain$.

Toujours à cause des problèmes de dépassement de mémoire, il n'est pas envisageable d'utiliser l'heuristique $Min\frac{Domain}{wdeg}$. L'heuristique $Min\frac{Domain}{wvar}$ est moins coûteuse mais ne semble pas assez riche pour prendre en compte les contraintes n-aires. Nous avons donc développé un nouvel apprentissage dénommé weighted culprit variable (wcvar) inspiré du wdeg et du wvar. Comme le wvar, le wcvar associe un poids à chaque variable. Ce poids est incrémenté chaque fois qu'une variable est responsable du dernier conflit ayant provoqué un échec.

Cet apprentissage est utilisé pour l'ordonnancement dynamique, spécifique et adaptatif (HDSA_WCVar) :

- des variables n'ayant pas la valeur 0 dans leur domaine, en favorisant les plus contraintes : $Min\frac{Domain}{wcvar}$,
- des autres variables présentant la valeur 0 dans leur domaine, en favorisant les moins contraintes : $Max\frac{Objectif}{wcvar}$.

Heuristique basée sur le dernier conflit Dans un deuxième temps, une heuristique dynamique, spécifique et adaptative basée sur le dernier conflit (HDSA_LC) a été implémentée. Le raisonnement à partir du dernier conflit [16] consiste, en cas de retour-arrière, à affecter en priorité la dernière variable en échec. L'heuristique HDSA_LC résultante est :

1. la dernière variable en échec est affectée d'abord,
2. les variables ayant une seule valeur dans leur domaine sont affectées ensuite,

3. les variables n'ayant pas la valeur 0 dans leur domaine suivant $MinDomain$,
4. les variables n'ayant pas la valeur 0 dans leur domaine et ayant la même taille de domaine suivant $MinCompatible$,
5. les variables ayant la valeur 0 dans leur domaine suivant $MaxObjectif$,
6. les variables ayant la valeur 0 dans leur domaine et ayant le même impact sur le critère suivant $MaxCompatible$.

Combinaison Cette deuxième heuristique adaptative peut être combinée avec la première. Cette combinaison (HDSA_WCVar_LC) implique l'ordonnancement suivant :

1. la dernière variable en échec est affectée d'abord,
2. les variables ayant une seule valeur dans leur domaine sont affectées ensuite,
3. les variables n'ayant pas la valeur 0 dans leur domaine suivant $Min\frac{Domain}{wcvar}$,
4. les variables n'ayant pas la valeur 0 dans leur domaine et ayant la même taille de domaine suivant $MinCompatible$,
5. les variables ayant la valeur 0 dans leur domaine suivant $Max\frac{Objectif}{wcvar}$,
6. les variables ayant la valeur 0 dans leur domaine et ayant le même impact sur le critère suivant $MaxCompatible$.

4.3 Mécanismes de Restart

Malgré l'utilisation de mécanismes de retour-arrière intelligent et d'heuristiques dynamiques, spécifiques et adaptatives, les premières affectations restent déterminantes du fait du très grand nombre de variables. Les mécanismes de Restart permettent de remettre en cause ces premières affectations. Après un certain nombre de retour-arrière, la recherche est relancée depuis le début. Restart après restart, le nombre de retour-arrières autorisés croît de façon géométrique [22]. Le nombre croissant de retour-arrières autorisés garantit la complétude de l'algorithme. Cependant, il faut s'assurer de ne pas reparcourir la même partie de l'arbre de recherche. Pour cela il suffit de modifier l'ordonnancement dynamique des variables. Plusieurs types de Restart ont été développés.

Le Restart bruité (Rs_Rand) bruite les heuristiques d'ordonnancement [9]. Pour cela, la valeur heuristique des variables est bruitée aléatoirement de plus ou moins un certain pourcentage. En plus d'assurer un parcours d'arbre différent, ce bruitage permet une exploration aléatoire autour de l'heuristique initiale. Il est donc possible de bruiter les quatre heuristiques HDS, HDSA_WCVar, HDSA_LC et HDSA_WCVar_LC pour obtenir quatre Rs_Rand différents. Le HDS_Rs_Rand est un Restart bruité, les trois autres sont des Restarts adaptatifs bruités.

Le Restart adaptatif avec utilisation de l'apprentissage wdeg a été introduit par D. Grimes [10]. Dans son travail, il a montré l'intérêt de modifier l'ordonnancement uniquement grâce à l'apprentissage wdeg. Un Restart adaptatif similaire est proposé sur la base de l'heuristique wcvar (HDSA_WCVar_Rs). HDSA_WCVar_Rs n'utilise aucun mécanisme de bruitage. Seul l'apprentissage wcvar assure un parcours d'arbre différent après chaque restart. Un Restart adaptatif sans bruitage à base d'une combinaison de wcvar et de dernier conflit (HDSA_WCVar_LC_Rs) a également été développé.

L'ensemble de ces mécanismes de retour-arrière, d'ordonnancement et de restart ont été expérimentés sur des instances correspondant aux satellites en cours de conception.

5 Expérimentations

Les expérimentations ont été menées sur cinq charges utiles de télécommunication associées chacune à une base d'exigences, appelées ici SatA, SatB, SatD, SatE et SatF. Afin d'assurer une grande variété d'instances de test, plusieurs dizaines d'instances ont été dérivées aléatoirement de ces instances de base. Pour cela, les exigences de tests prises en compte par chaque instance ont été tirées aléatoirement à partir de chaque base d'exigences. Les instances de test de type SatA et SatB impliquent plusieurs centaines de variables, celles de types SatD, SatE et SatF en comptent plusieurs milliers. Sur ces instances de test, douze algorithmes ont été comparés. Tous sont à base de recherche en profondeur d'abord et de propagation par Forward Checking. Ils se différencient par les mécanismes de retour-arrière (§4.1), les heuristiques (§4.2) et les mécanismes de Restart (§4.3) utilisés. Ces douze algorithmes sont les suivants :

- Backtrack et heuristique dynamique et spécifique (BT_HDS),
- Backjumping et heuristique dynamique et spécifique (BJ_HDS),
- Conflict Directed Backjumping (CBJ) et heuristique dynamique et spécifique (CBJ_HDS).
- CBJ et heuristique dynamique, spécifique et adaptative avec le wcvar (CBJ_HDSA_WCVar),
- CBJ et heuristique dynamique, spécifique et adaptative avec le dernier conflit (CBJ_HDSA_LC),
- CBJ et heuristique dynamique, spécifique et adaptative avec le wcvar et le dernier conflit

(CBJ_HDSA_WCVar_LC),
- CBJ, heuristique dynamique et spécifique et Restart bruité (CBJ_HDS_Rs_Rand),
- CBJ, heuristique dynamique, spécifique et adaptative avec le wcvar et Restart (CBJ_HDSA_WCVar_Rs),
- CBJ, heuristique dynamique, spécifique et adaptative avec le wcvar et Restart bruité (CBJ_HDSA_WCVar_Rs_Rand),
- CBJ, heuristique dynamique et spécifique et adaptative avec le dernier conflit et Restart bruité (CBJ_HDSA_LC_Rs_Rand),
- CBJ, heuristique dynamique et spécifique et adaptative avec le wcvar et le dernier conflit et Restart (CBJ_HDSA_WCVar_LC_Rs),
- CBJ, heuristique dynamique et spécifique et adaptative avec le wcvar et le dernier conflit et Restart bruité (CBJ_HDSA_WCVAR_LC_Rs_Rand).

Ces douze algorithmes ont également été comparés au solveur CP Optimizer d'IBM Ilog [1] pour les instances de type SatA, SatB et SatE n'ayant pas provoqué de dépassement de mémoire. Ce solveur générique a été utilisé en mode :
- profondeur d'abord (CP_Optimizer),
- Restart (CP_Optimizer_Rs).

Pour chaque instance et chaque algorithme, on mesure le nombre d'exigences de test satisfaites du meilleur plan de test produit en temps borné : une minute pour les instances simples de type SatA et SatB, cinq minutes pour les autres. Pour chaque type d'instance et chaque algorithme, une moyenne de ces mesures est réalisée. Ces moyennes sont normalisées entre 0 et 1 pour fournir un indice de qualité. Ces normalisations sont effectuées à partir des algorithmes spécifiques puisque CP Optimizer ne produit pas de résultats pour les instances de type SatD et SatE. Ainsi, le meilleur algorithme a une note de 1 et le plus mauvais de 0. Les indices de qualité des solutions trouvées par CP Optimizer sont également calculés par rapport aux solutions des algorithmes spécifiques. Pour cette raison, il est possible d'avoir un indice supérieur à 1 si la solution trouvée par CP_Optimizer est meilleure que celles produites par les algorithmes spécifiques, inférieur à 0 si elle est moins bonne. Le tableau 1 regroupe les indices de qualités obtenus par les douze algorithmes spécifiques et par le solveur générique pour chaque type d'instance. La figure 2 représente la moyenne de ces indices de qualité sur les cinq types d'instances pour les algorithmes spécifiques.

Résultats des mécanismes de retour-arrière Le tableau 1 montre que le BackJumping (BJ_HDS) permet d'améliorer très légèrement la qualité des solutions trouvées par rapport au Backtrack (BT_HDS). On note jusqu'à $0,43$ de gain d'indice pour les instances de type SatE. Pour la plupart des types d'instances de test, le CBJ (CBJ_HDS) apporte une amélioration importante à la qualité des solutions produites. Jusqu'à 0.63 de gain est obtenu pour les instances de type SatE par rapport au BT_HDS. La figure 2 confirme ces résultats. Bien que le CBJ_HDS soit le plus coûteux de ces mécanismes de retour-arrière, il permet d'obtenir un meilleur indice moyen que BJ_HDS et le BT_HDS pour le même temps d'exécution. Pour cette raison, les heuristiques et les Restarts ont été expérimentés uniquement sur la base du CBJ.

Résultats des mécanismes d'heuristiques d'ordonnancement des variables Les solutions trouvées par l'heuristique adaptative wcvar sans Restart (CBJ_HDSA_WCVar) ne sont pas meilleures que celles obtenues par l'heuristique spécifique (CBJ_HDS) (voir tableau 1 et figure 2), à l'exception d'un léger gain pour les instances de type SatF. L'heuristique dirigé par le dernier conflit (CBJ_HDSA_LC) permet un léger gain, $0,02$, pour les instances de type SatE par rapport à l'heuristique spécifique. L'heuristique adaptative combinée (CBJ_HDSA_WCVar_LC) détériore de près de $0,20$ en moyenne les solutions par rapport à CBJ_HDS. Pour les instances de type SatA, cette heuristique mixte dégrade même les solutions par rapport au BT_HDS. Ces apprentissages ne semblent donc pas être utilisés à bon escient sans remise en cause des premières affectations.

Résultats des mécanismes de Restart Parmi les algorithmes spécifiques, le tableau 1 montre que les meilleurs résultats sont obtenus :
- pour les instances de type SatA, par le Restart adaptatif bruité dirigé par le dernier conflit (CBJ_HDSA_LC_Rs_Rand),
- pour les instances de type SatB, à égalité par le Restart bruité (CBJ_HDS_Rs_Rand) et le Restart adaptatif bruité dirigé par le dernier conflit (CBJ_HDSA_LC_Rs_Rand),
- pour les instances de type SatD, SatE et SatF, par le Restart adaptatif wcvar (CBJ_HDSA_WCVar_Rs).

En moyenne, le Restart adaptatif wcvar bruité (CBJ_HDSA_WCVar_Rs_Rand) et le Restart adaptatif wcvar bruité dirigé par le dernier conflit (CBJ_HDSA_WCVar_LC_Rs_Rand) ne permettent pas d'améliorer les résultats obtenus par le CBJ_HDS (voir figure 2). Ces Restarts dégradent même les résultats obtenus par le BT_HDS pour les instances de type SatA. En moyenne, l'utilisation du dernier conflit (CBJ_HDSA_LC_Rs_Rand ou respecti-

Algorithme	**SatA**	**SatB**	**SatD**	**SatE**	**SatF**
BT_HDS	0,76	0	0	0	0,46
BJ_HDS	0,76	0,03	0,01	0,43	0,47
CBJ_HDS	0,81	0,14	0,08	0,63	0,48
CBJ_HDSA_WCVar	0,81	0,09	0,08	0,51	0,49
CBJ_HDSA_LC	0,81	0,14	0,08	0,65	0,48
CBJ_HDSA_WCVar_LC	0	0,07	0,08	0,53	0,49
CBJ_HDS_Rs_Rand	0,93	1	0,53	0,79	0,05
CBJ_HDSA_WCVar_Rs	0,9	0,74	1	1	1
CBJ_HDSA_WCVar_Rs_Rand	0,13	0,71	0,08	0,55	0,15
CBJ_HDSA_LC_Rs_Rand	1	1	0,44	0,86	0
CBJ_HDSA_WCVar_LC_Rs	0,9	0,72	0,97	0,99	0,99
CBJ_HDSA_WCVar_LC_Rs_Rand	0,13	0,71	0,08	0,52	0,14
CP_Optimizer	-23,5	-8,47	NA	NA	-17,94
CP_Optimizer_Rs	0,36	1,26	NA	NA	0,7

Légende :

Les cellules grisées NA correspondent à des tests non applicables dus à des problèmes de dépassement de mémoire.

TABLEAU 1 – Indice de qualité de chaque algorithme pour chaque type d'instances.

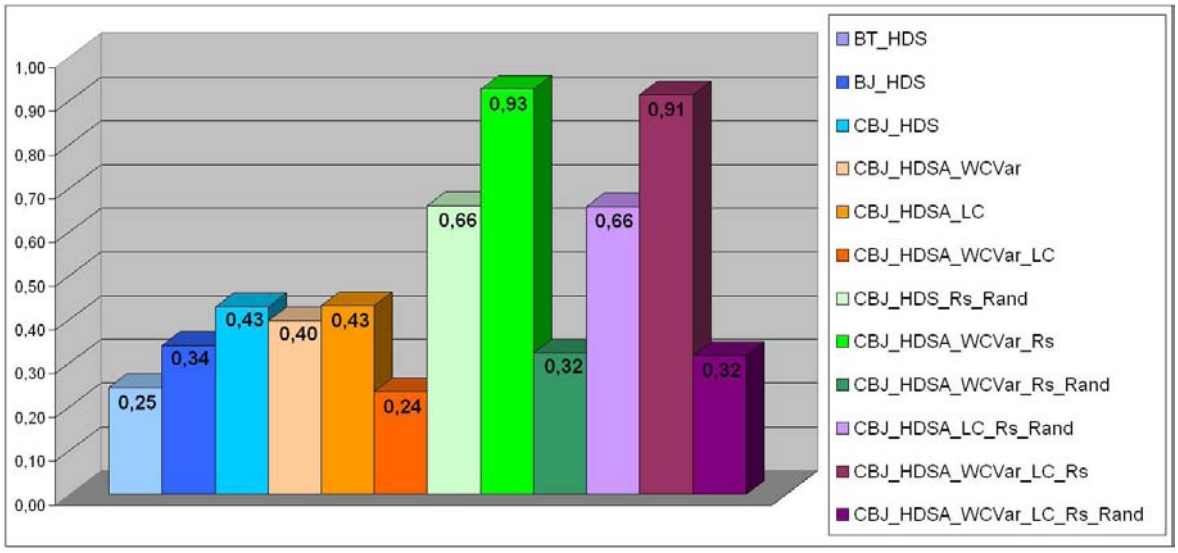


FIGURE 2 – Indice moyen de qualité pour chaque algorithme spécifique.

vement CBJ_HDSA_WCVar_LC_Rs) dégrade le restart bruité (CBJ_HDS_Rs_Rand) ou respectivement le restart adaptatif wcvar (CBJ_HDSA_WCVar_Rs). Bien qu'en combinaison avec le restart bruité (CBJ_HDSA_LC_Rs_Rand), elle permet d'obtenir les meilleurs résultats pour les instances de type SatA et SatB. En moyenne, le Restart adaptatif wcvar (CBJ_HDSA_WCVar_Rs) permet d'obtenir les meilleurs indices de qualité.

Cette analyse permet d'affirmer l'intérêt des mécanismes de CBJ et de Restart pour ce problème d'optimisation des plans de test avec un net avantage pour le Restart adaptatif wcvar. Cet apprentissage wcvar basé sur les variables responsables des échecs, s'avère très fructueux lors de l'utilisation du Restart.

Résultats du solveur générique : CP Optimizer Le solveur générique CP Optimizer d'IBM Ilog [1] a été utilisé à titre indicatif sur les instances de types SatA, SatB et SatF. Il n'a pas pu être utilisé sur les autres instances du fait de problème de dépassement de mémoire. Le tableau 1 montre que CP Optimizer en profondeur d'abord (CP_Optimizer) dégrade énormément l'ensemble des plans de test. Comme nos algorithmes utilisent le même mécanisme de recherche en profondeur d'abord, ce résultat montre l'impact positif du CBJ et des heuristiques spécifiques développés pour ce problème. Le Restart de CP Optimizer (CP_Optimizer_Rs) est plus performant. Il permet de trouver de meilleurs solutions pour les instances de type SatB. Par contre, il est très loin des solutions optimales pour les instances de type SatA, plus de 0.60

d'écart d'indices. Pour obtenir de meilleurs résultats pour les instances de type SatB, il serait intéressant d'implémenter d'autres mécanismes de Restart adaptatif basés sur l'apprentissage de la réduction de l'arbre de recherche. Ce type d'apprentissage proposé par P. Refalo [19] consiste à apprendre l'impact de chaque affectation sur la taille de l'arbre de recherche.

6 Conclusion

Pour ce nouveau domaine applicatif, le modèle mathématique que nous avons proposé permet de représenter le besoin d'optimisation des plans de test des charges utiles des satellites de télécommunication. Cependant, il engendre un très grand nombre de variables (plusieurs milliers) et de contraintes (plusieurs centaines de milliers). Sa résolution par des solveurs génériques peut s'avérer impossible.

Seuls les algorithmes spécifiques implémentés ont permis la résolution de toutes les instances de test. Au sein de ces algorithmes, les mécanismes de CBJ et de Restart se sont révélés bénéfiques. Pour les instances les plus complexes, la nouvelle heuristique wcvar s'est révélée très performante en combinaison avec le Restart. Pour un type d'instances simples, les Restarts mis en place ne sont pas aussi performants que CP Optimizer en mode Restart. Il semblerait intéressant de développer un apprentissage basé sur l'impact des affectations sur la taille de l'arbre de recherche [19].

Ce problème d'optimisation pourrait par ailleurs être traité par des algorithmes à base de recherche locale tels que le Min Conflict [17], la recherche Tabou [8] ou encore le Recuit Simulé [15].

Références

[1] IBM ILOG OPL Development Studio. http ://www-01.ibm.com/software/integration/optimization/opl-dev-studio/.

[2] C. Bessière and J.C. Régin. MAC and Combined Heuristics : Two Reasons to Forsake FC (and CBJ ?) on Hard Problems. In *CP*, pages 61–75, 1996.

[3] J. R. Bitner and E. M. Reingold. Backtrack Programming Techniques. *Commun. ACM*, 18(11) :651–656, 1975.

[4] F. Boussemart, F. Hemery, C. Lecoutre, and L. Sais. Boosting Systematic Search by Weighting Constraints. In *ECAI*, 2004.

[5] R. Dechter. *Constraint Processing*. Morgan Kaufmann Publishers Inc., 2003.

[6] L. Epstein, A. Levin, and R. van Stee. Multidimensional Packing with Conflicts. In *FCT*, pages 288–299, 2007.

[7] J. G. Gaschnig. *Performance measurement and analysis of certain search algorithms*. PhD thesis, 1979.

[8] F. Glover and M. Laguna. Tabu Search. In *Modern Heuristic Techniques for Combinatorial Problems*, pages 70–141. 1993.

[9] C.P. Gomes, B. Selman, and H. Kautz. Boosting combinatorial search through randomization. In *AAAI*, pages 431–437, 1998.

[10] D. Grimes. A Study of Adaptive Restarting Strategies for Solving Constraint Satisfaction Problems. In *AICS*, 2008.

[11] C. Guéret, C. Prins, and M. Sevaux. *Programmation linéaire*. Eyrolles, 2000.

[12] R. M. Haralick and G. L. Elliott. Increasing Tree Search Efficiency for Constraint Satisfaction Problems. *Artificial Intelligence*, 14(3) :263–313, 1980.

[13] W. Karoui. *Méthodes à divergences pour la résolution de problèmes de satisfaction de contraintes et d'optimisation combinatoire*. PhD thesis, 2010.

[14] H. Kellerer, U. Pferschy, and D. Pisinger. *Knapsack Problems*. Springer, Berlin, Germany, 2004.

[15] S. Kirkpatrick, C.D. Gelatt, and M.P. Vecchi. Optimization by Simulated Annealing. *Science*, 220(4598) :671–680, 1983.

[16] C. Lecoutre, L. Saïs, S. Tabary, and V. Vidal. Reasoning from Last Conflict(s) in Constraint Programming. *AIJ*, 173 :1592–1614, 2009.

[17] S. Minton, M. D. Johnston, A. B. Philips, and P. Laird. Solving large-scale constraint satisfaction and scheduling problems using a heuristic repair method. In *AAAI*, pages 17–24, 1990.

[18] P. Prosser. Hybrid Algorithms for the Constraint Satisfaction Problem. *Computational Intelligence*, 9 :268–299, 1993.

[19] P. Refalo. Impact-Based Search Strategies for Constraint Programming. In *CP*, volume 3258, pages 557–571, 2004.

[20] D. Sabin and E. C. Freuder. Contradicting conventional wisdom in constraint satisfaction. In *ECAI*, pages 125–129, 1994.

[21] T. Schiex and G. Verfaillie. Nogood Recording for Static and Dynamic Constraint Satisfaction Problems. In *ICTAI*, 1993.

[22] T. Walsh. Search in a Small World. In *IJCAI*, pages 1172–1177, 1999.

Heuristiques Adaptatives Génériques pour la Recherche en Grand Voisinage

Jean-Baptiste Mairy[1], Pierre Schaus[2] et Yves Deville[1]

[1]Université catholique de Louvain, Belgique
[2]n-Side, Louvain-la-Neuve,Belgique
[1]{jean-baptiste.mairy,yves.deville}@uclouvain.be [2]pierre.schaus@n-side.com

Abstract

Le paradigme de Recherche en Grand Voisinage (LNS) a d'ores et déjà prouvé son efficacité pour la résolution d'une grande variété de Problèmes d'Optimisation Contraints (COP). Le LNS est une métaheuristique de recherche locale (LS) utilisant une recherche exhaustive (telle que CP) pour explorer les voisinages. Ces voisinages sont définis en relaxant un sous-ensemble des variables du problème (appelé fragment) par rapport à la solution courante. Malgré le succès du LNS, il n'existe aucun principe général pour choisir les fragments à relaxer. En l'absence d'une bonne connaissance du problème, ceux-ci sont généralement choisis aléatoirement pour favoriser la diversification. Ce papier [1] propose différentes heuristiques pour un choix automatique et générique de fragments améliorant le choix aléatoire. Ces heuristiques sont testées sur le problème du Car Sequencing pour lequel une modélisation originale est introduite. Ces expériences permettent de montrer que les heuristiques proposées sont plus performantes qu'un choix aléatoire. Deux d'entre elles (impact dynamique proximité et impact dynamique max proximité) sont également significativement meilleures que toutes les autres.

1 Introduction

L'obtention de solutions optimales pour les problèmes d'optimisation combinatoire sous contraintes est un des chalenges majeures pour les sciences informatiques. Le paradigme de Recherche en Grand Voisinage (LNS) s'est montré efficace dans bon nombre de problèmes d'optimisation sous contraintes, en permettant l'obtention rapide de bonnes solutions (par exemple [1, 3, 4, 5, 6]). Cette efficacité vient de la combinaison de l'expressivité de la programmation par contraintes (CP) avec la rapidité de la recherche locale (LS). Le principe de fonctionnement du LNS est de maintenir, durant toute la recherche, une solution (appelée solution courante) qui ne viole aucune contrainte mais qui n'est pas optimale. Cette solution courante correspond à la meilleure solution trouvée depuis le début de la recherche. Les solutions successives sont obtenues en répétant les 2 opérations suivantes jusqu'à arriver à une condition d'arrêt :

1. définition du voisinage : cette étape consiste à choisir le sous-ensemble des variables qui seront relâchées à leur domaine initial, les autres étant affectées à leur valeur dans la solution courante. Le voisinage de la solution courante est, dès lors, défini par l'ensemble des solutions du sous-problème ainsi défini.
2. exploration du voisinage : CP est utilisé dans cette étape pour explorer le sous-problème défini à l'étape 1. Lorsqu'une solution meilleure est trouvée, elle devient la solution courante. Afin d'éviter de passer trop de temps à explorer les voisinages, cette étape est limitée par une borne sur le temps ou le nombre d'échecs de la recherche.

En dépit de la simplicité de cette métaheuristique, la procédure de sélection des fragments doit être choisie précautionneusement. Malheureusement, il n'existe pas de principe général pour réaliser cette tâche. En l'absence d'une connaissance accrue du problème traité, les fragments sont souvent choisis aléatoirement dans le but de favoriser la diversification.

P. Laborie et D. Godard proposent, dans [8], un LNS adaptatif pour les problèmes de scheduling. Les trois stratégies proposées sont combinées en un portfolio. À chaque étape de sélection d'un fragment, une des stratégie est choisie en fonction de probabilités pour les

1. Ce papier a été publié dans sa version anglaise au workshop LSCS de la conférence CP2010

techniques d'être utilisées. Ces probabilités sont ensuite mises à jour en fonction des résultats de l'exploration du voisinage. Malheureusement, les trois stratégies utilisées sont spécifiques au problème du scheduling.

L. Perron et al. introduisent, dans [1], deux heuristiques adaptatives génériques de sélection de fragments. La première stratégie démarre du CSP original (où toutes les variables ont pour domaines leur domaine initial) et sélectionne successivement les variables à fixer pour définir le voisinage. Quand une variable est sélectionnée, elle est fixée à sa valeur dans la solution courante. La propagation de cette assignation est réalisée. L'heuristique sélectionne alors, comme variable suivante, la variable dont le domaine a été le plus réduit par cette assignation. La seconde heuristique sélectionne les variables à relâcher. Le but de cette heuristique est de définir des fragments composés de variables proches. La relation de proximité est calculée par rapport à la propagation ayant eu lieu lors des étapes précédentes d'assignation des variables ne faisant pas partie des fragment.

Ce papier poursuis le travail de Perron et al. en apportant les contributions suivantes :

- De nouvelles heuristiques adaptatives génériques de sélection de fragments
- Une modélisation originale du problème du Car Sequencing comme un problème d'optimisation contraint
- Des résultats expérimentaux montrant l'efficacité de ces heuristiques sur le problème du Car Sequencing

La suite du papier est organisée comme suit : la section 2 définit les heuristiques de sélection de fragments et la section 3 présente les résultats des expérimentations de ces heuristiques sur le problème du Car Sequencing et la section 4 conclut cet article.

2 Heuristiques Adaptives Génériques de sélection de fragments

Cette section se concentre sur les différentes heuristiques développées dans le cadre de ce travail. D'abord, les concepts et les mesures utilisés pour définir les heuristiques sont introduits. Ensuite, les heuristiques simples sont définies. Celles-ci n'utilisent qu'une mesure afin de sélectionner les fragments. Finalement, les heuristiques combinées, utilisant plus d'une mesure, sont présentées.

2.1 Concepts et mesures

Les différents concepts et mesures présentés dans cette section sont : l'impact dynamique, l'impact dynamique moyen, l'impact dynamique max, l'impact adaptatif et la proximité.

Impact dynamique. L'impact dynamique d'une variable tente de mesurer l'impact que la variable aurait sur l'objectif si elle était relaxée. Soit S une solution au problème, l'impact d'une valeur v pour une variable x est défini comme la différence qui est induite sur l'objectif par l'affectation de v à x à la place de sa valeur dans S. Pendant l'évaluation de cet impact, les autres variables sont considérées comme assignées à leur valeur dans S et les contraintes sont ignorées. Pratiquement, pour une fonction objectif O, l'impact dynamique de la valeur v pour la variable x_i par rapport à la solution S se calcule comme :

$$\mathcal{I}(x_i, v, S) = O\Big(S(x_1), \ldots, S(x_{i-1}), v, S(x_{i+1}), \ldots, S(x_n)\Big) - O\Big(S(x_1), \ldots, S(x_n)\Big),$$

où $S(x_k)$ représente la valeur de x_k dans S.

L'impact d'une variable x pour une solution S est l'agrégation des $\mathcal{I}(x, v, S)$ pour toutes les valeurs $v \in D_0(x)$ (domaine initial de x). L'impact dynamique défini ici est séparé en deux mesures correspondant chacune à une agrégation différente : l'impact dynamique moyen et l'impact dynamique max.

Impact dynamique moyen. Soit x, une variable, O la fonction objectif du problème et S, une solution. L'impact dynamique moyen de x par rapport à S est défini comme

$$\mathcal{I}_{mean}(x, S) = \frac{1}{|D_0(x)| - 1} \sum_{v \in D_0(x) \setminus \{S(x)\}} \mathcal{I}(x, v, S).$$

Une variable qui, en moyenne, améliore l'objectif lorsqu'on ne change que sa valeur par rapport à la solution S aura un $\mathcal{I}_{mean}$ positif si le problème est une maximisation et négatif si le problème est une minimisation.

Impact dynamique max. Pour les problèmes de maximisation, l'impact dynamique max est défini, pour une variable x, comme la valeur maximale de $\mathcal{I}(x, v, S)$ pour $v \in D_0(x) \setminus \{S(x)\}$. Cela correspond à l'amélioration maximum que la variable pourrait, à elle seule, induire sur l'objectif. Pour une variable x et une solution S, l'impact dynamique max est donc

$$\mathcal{I}_{max}(x, S) = \max_{v \in D_0(x) \setminus \{S(x)\}} \mathcal{I}(x, v, S).$$

Pour les problèmes de minimisation, $\mathcal{I}_{min}$ est défini de manière analogue.

Impact Adaptatif. L'impact adaptatif tente de saisir l'impact des variables sur la fonction objectif en utilisant les informations révélées par la propagation, produites lors des affectations. L'objectif est de refléter l'importance intrinsèque des variables pour l'objectif, indépendamment de la solution actuelle. Plus précisément, l'impact adaptatif mesure l'influence que les affectations des variables ont sur les bornes de l'objectif par le biais de la propagation. Pour les problèmes de maximisation, les bornes utilisées sont les bornes supérieures. Pour les minimisations, ce sont les bornes inférieures. L'impact adaptatif d'une variable x est défini comme la moyenne des variations de la borne d'intérêt sur toutes les affectations de x, depuis le début de la recherche. Supposons que le LNS soit à un point d'avancement p et que $SumImpact(x, p)$ soit la somme des changements de la borne d'intérêt qui ont eu lieu lors des affectations de x jusqu'au point p. Supposons également que $NumTimesBound(x, p)$ représente le nombre de fois où x a été lié jusqu'au point d'avancement p. L'impact adaptatif de x est alors :

$$\mathcal{I}_{adaptive}(x, p) = \frac{SumImpact(x, p)}{NumTimesBound(x, p)}.$$

Cette définition d'impact adaptatif traite chaque variable indépendamment des autres. De futures définitions pourront inclure les interactions entre variables.

Proximité. La proximité est une mesure qui a pour but de caractériser les relations entre les variables. Cette mesure a été inspirée par celle définie dans [1]. La proximité est définie en termes d'interactions que le réseau de contraintes induit entre deux variables. Elle se mesure en inspectant la propagation se produisant lorsque les variables sont liées. Pratiquement, la proximité entre les variables x et y est calculée comme le nombre moyen de valeurs qui sont retirées de $D(y)$ lorsque x est affecté. La seule subtilité est que cette moyenne est calculée uniquement sur les affectations de x où y n'est pas lié. Sans cette subtilité, le calcul inclurait les cas où aucune valeur de y n'est retirée uniquement parce que cette variable a déjà été affectée.

Supposons que le LNS soit à un point p, où $Propag(x, y, p)$ est le nombre total de valeurs supprimées de $D(y)$ par toutes les affectations du x jusqu'à p. Soit également $NumBinds(x, y, p)$ le nombre de fois où x a été lié alors que y n'était pas affecté jusqu'à p. La proximité entre x et y est alors définie comme

$$Proximity(x, y, p) = \frac{Propag(x, y, p)}{NumBinds(x, y, p)}.$$

Si deux variables ont une grande proximité, la relaxation d'une de ces variables sans l'autre retirerait, avant le début de la recherche, un grand nombre de valeurs du domaine de la variable relâchée. Cela réduirait les possibilités de changement de valeur pour la variable relâchée et donc, les possibilités de réoptimisation de l'objectif. Notez que la relation de proximité n'est pas symétrique.

La mesure de proximité définie ici est basée sur la même intuition que la relation de proximité définie dans [1] : les relations entre les variables peuvent être révélées par la propagation. Cependant, la proximité définie dans ce papier est mise à jour chaque fois qu'une variable est liée. Cela inclut le gel des variables ne faisant pas partie des fragments et les points de choix dans les étapes de recherche CP. La mesure de proximité définie dans [1] est seulement mise à jour au cours du gel des variables hors fragments. En outre, comme nous le verrons, la proximité est ici uniquement utilisée en combinaison avec d'autres heuristiques, ce qui n'est pas le cas dans [1].

```
fragment = {}
varSet = X
while (|fragment| < k) do
  Select variable x in varSet according to I_λ
  fragment ⊕ x
  varSet ⊖ x
end while
```

Pseudo-code 2.1 :
Sélection de fragment basée sur l'impact

2.2 Heuristiques simples

Cette section présente les heuristiques simples. Chacune de ces heuristiques utilise une des mesures déjà présentées.

Ces heuristiques ont pour objectif de sélectionner un fragment composé de variables ayant un impact important sur l'objectif. Pour favoriser la diversification, la sélection est randomisée. En effet, les variables à inclure dans les fragments sont sélectionnées *parmi* les variables d'impact maximum. Il existe une heuristique simple pour chaque mesure d'impact définie dans la section 2

Le pseudo-code 2.1 montre le processus de sélection des fragments. Dans ce code, X désigne l'ensemble des variables du problème, k correspond à la taille du fragment à sélectionner, $\mathcal{I}_\lambda$ est la mesure d'impact choisie, $\oplus$ et $\ominus$ sont respectivement l'ajout et la suppression d'un élément dans une liste. La sélection des variables est aléatoire, avec une distribution de probabilité proportionnelle à $\mathcal{I}_\lambda$.

2.3 Heuristiques combinées

Cette section présente les heuristiques combinées. Chacune de ces heuristiques combine une mesure d'impact avec celle de proximité. Il y a donc une heuristique combinée pour chaque notion d'impact définie.

Dans ces heuristiques, le fragment est sélectionné en deux parties, chacune ayant une fonctionnalité distincte. Le but est de sélectionner des fragments composés de variables importantes (1re partie) et des variables qui sont liées avec les plus importantes (2e partie). Cette seconde partie du fragment permet aux variables importantes de changer de valeur pour améliorer la solution courante.

L'impact et la proximité sont ici considérés comme également pertinents. Le fragment est donc séparé en deux parties égales. Dans la seconde partie du fragment, pour déterminer si une variable est liée ou non à une autre, un seuil sur la proximité est utilisé. Une variable x est considérée comme liée à une autre variable y si $proximity(x, y)$ est supérieur au seuil. De plus, la proximité n'étant pas symétrique, elle est utilisée dans les deux sens. Pour une variable x, une première moitié de fragment F_1 et un seuil T, son inclusion dans la seconde moitié du fragment est déterminée par la somme du nombre de variables de F_1 auquel x est lié et du nombre de variables de F_1 qui sont liées à x :

$$NL(x_i, F_1, T) = \big|\{y \in F_1 \mid proximity(x, y) > T\}\big| + \big|\{y \in F_1 \mid proximity(y, x) > T\}\big|$$

L'algorithme pseudo-code 2.2 montre comment les fragments sont sélectionnés par les heuristiques de cette section. Dans ce code, $\mathcal{I}_\lambda$ est la mesure d'impact choisie. La sélection de la première moitié du fragment (F_1) est la même que celle présentée pour les heuristiques basées sur l'impact, sauf pour le nombre de variables. La diversification est assurée dans les deux parties du fragment par le même mécanisme aléatoire que pour les heuristiques précédentes.

```
F1 = {}
varSet = X
while (|F1| < k/2) do
  Select variable x in varSet according to I_λ
  F1 ⊕ x
  varSet ⊖ x
end while
F2 = {}
while (|F2| < k/2) do
  Select variable x in varSet among large NL(x, F1, T)
  F2 ⊕ x
  varSet ⊖ x
end while
fragment = F1 ∪ F2
```

Pseudo-code 2.2 :
Sélection de fragment basée sur l'impact et la proximité

3 Résultats expérimentaux

Cette section présente les résultats expérimentaux. Elle commence par présenter le problème utilisé pour les tests. Ensuite, elle introduit la méthodologie utilisée. Finalement, elle présente les résultats des expériences et leur interprétation.

3.1 Le Car Sequencing comme un problème d'optimisation

Le problème choisi pour tester les différentes heuristiques présentées dans ce papier est le problème du Car Sequencing. Cette section décrit ce problème ainsi que sa modélisation comme un problème d'optimisation sous contraintes.

Le problème du Car Sequencing consiste à ordonner des voitures sur une ligne de production pour satisfaire une demande. Les caractéristiques des voitures à produire sont regroupées en configurations. Une configuration est composée d'options spécifiques (telles que toit ouvrant, climatisation, etc.). Chaque option o est installée par une station qui a une capacité limitée : la station peut installer o sur au plus m_o voitures dans toute séquence de longueur n_o. Le but de ce problème est d'ordonner les voitures à produire de sorte que les capacités des stations ne soient pas dépassées.

Ce problème de faisabilité a été ici modélisé comme un problème d'optimisation sous contraintes en relâchant la contrainte imposant qu'un emplacement de la ligne de production ne puisse contenir qu'une voiture. Plusieurs voitures de la même configuration peuvent donc être situées sur le même emplacement. L'objectif du problème relaxé est de maximiser le nombre d'emplacements utilisés sur la ligne de production (et donc minimiser le nombre d'emplacements avec plu-

sieurs voitures). Une solution pour le problème initial (problème de satisfaction) correspond à une solution du problème relaxé où le nombre d'emplacements utilisés est égal au nombre de voitures. Cette relaxation est différente de la relaxation usuelle (minimisant le nombre de violations des contraintes de capacité) parce que le LNS ne considère que des solutions sans viol de contraintes.

La Figure 1 présente le modèle *Comet* pour ce problème. Les données du problème sont : *demand*, le nombre de voitures à produire pour chaque configuration ; m et n les contraintes de capacité pour chaque option et $configOfOption$, le tableau contenant pour chaque option, l'ensemble des configurations qui la contiennent. Dans ce modèle, $carsWithConfig$ est un tableau contenant les voitures de chaque configuration. Les variables de décision sont contenues dans $placeOfCar$, représentant l'emplacement où chaque voiture est placée sur la ligne de production. Le tableau de variables $configOfCar$ contient la configuration des voitures présentes sur les emplacements de la ligne. Il permet d'utiliser la contrainte *sequence*. Les deux points de vue ($placeOfCar$ et $configOfCar$) sont liés par des contraintes *element*. Il est important de remarquer qu'un emplacement vide (c'est à dire un emplacement non utilisé dans $placeOfCars$) peut contenir n'importe quelle configuration de voiture qui est compatible avec les configurations des voitures environnantes. Cette contrainte permet une réduction importante de l'espace de recherche. Nous pensons donc que cette relaxation est plus forte que celle proposée dans [1]. Dans cette relaxation, des emplacements supplémentaires et des voitures virtuelles ne nécessitant pas d'option sont introduits. L'objectif est alors de minimiser le dernier emplacement avec une voiture réelle. La relaxation présentée ici permet d'utiliser la contrainte globale $atLeastNValue$ [7], qui est une version soft de la contrainte bien connue $AllDifferent$. Cette contrainte lie globalement l'objectif (le nombre d'emplacements non-vides de la ligne de production) et les variables de décision, permettant un filtrage fort. Un autre avantage de $atLeastNValue$ est que le couplage maximal maintenu par son propagateur peut servir d'heuristique de valeur pour affecter les voitures aux emplacements. Nous pouvons retrouver cette heuristique dans l'alternative gauche du bloc *using* du modèle *Comet*. L'inconvénient de ce modèle est qu'il introduit des symétries entre les voitures de même configuration. Ces symétries sont brisées dynamiquement au cours de la recherche dans l'alternative de droite du bloc *using* : en entrant dans l'alternative de droite, il est prouvé que l'emplacement est impossible pour la voiture et est donc supprimé des domaines de toutes les voitures non assignées de même configuration. Chaque fois que la limite sur le nombre d'échecs est atteinte ou que l'arbre de recherche est exploré en entier, le bloc *onRestart* est exécuté, un fragment est choisi et les variables n'appartenant pas à ce fragment sont fixées à leur valeur dans la solution courante (meilleure solution).

3.2 Méthodologie de test

Cette section présente la méthodologie utilisée pour la comparaison des différentes heuristiques présentées dans cet article. Les instances utilisées pour la partie expérimentale proviennent de la CSPLIB [2], seconde série. Toutes ces instances sont satisfaisables, ce qui signifie que, pour chacune d'elles, il existe une solution pour laquelle le nombre d'emplacements non-vides sur la ligne de production est égal au nombre de voitures.

La procédure de test a été conçue pour évaluer les heuristiques de sélection de fragments indépendamment des autres paramètres du LNS. Elle est basée sur la comparaison des résultats des stratégies sur plusieurs exécutions d'une seule étape LNS (sélection de fragment et recherche CP). Afin d'éliminer l'influence des paramètres, les valeurs optimales de ceux-ci sont évaluées pour chaque heuristique. La procédure de test peut être résumée comme suit :

1. Pour chaque instance, cinq solutions de départ sont obtenues à l'aide d'un LNS pour un nombre déterminé d'étapes (le même nombre pour toutes les solutions). Ces solutions, en plus des valeurs des variables, contiennent toutes les quantités qui sont requises par les mesures adaptatives.
2. À partir de ces solutions de départ, la taille optimale du fragment est évaluée pour chaque heuristique.
3. À partir de chaque solution de départ, chaque heuristique est exécutée plusieurs fois pour une unique étape LNS avec les paramètres optimaux.

Pour l'évaluation des paramètres optimaux, les tailles de fragments de 25, 20, 15, 10, 5, 2 pourcents du nombre total de variables sont testées. La limite sur la recherche CP est fixée à 3200 backtracks. Ce dernier paramètre peut être fixé a priori puisque la comparaison ne se fait que sur une étape LNS. Pour chaque heuristique et chaque solution de départ, la taille du fragment utilisée pour les comparaisons est celle qui mène au plus grand nombre d'améliorations sur 10 exécutions.

2. `www.csplib.org`

```
//Data of the problem
range Options; range Configs; range Slots; range Cars;
int m[Options]; int n[Options]; //capacity m[o] out of n[o] for option o
int requires[Configs,Options];  //1 if config requires option, 0 otherwise
int demand[Configs]; //demand for each configuration c
int config[Cars]; //configuration for each car
set{int} configsOfOption[Options]; //set of cars with option
set{int} carsWithConfig[Configs]; //set of cars with config


Solver<CP> cp();
var<CP>{int} configOfCar[Slots](cp,Configs);//config of the car in the slot
var<CP>{int} placeOfCar[Cars](cp,Slots);//slot of the car
var<CP>{int} atLeast(cp,Cars);//number of slots with a car

AtLeastNValue<CP> atLeastCons;

cp.lnsOnFailure(3200); //LNS Limit

maximize<cp>
  atLeast
subject to{
  forall(o in Options)
    cp.post(sequence(configOfCar,demand,m[o],n[o],configsOfOption[o]));
  forall(c in Cars)
    cp.post(configOfCar[placeOfCar[c]]==config[c]);
  atLeastCons = atLeastNValue(placeOfCar,atLeast);
  cp.post(atLeastCons);
}
using{ //Search Heuristic
 while(!bound(placeOfCar))
    selectMin(c in Cars: !placeOfCar[c].bound())(placeOfCar[c].getSize()){
      int val = placeOfCar[c].getMin();
      if(atLeastCons.hasValInBestAssignment(c))
         val = atLeastCons.getValInBestAssignment(c);
      try<cp>{
         cp.post(placeOfCar[c]==val);
      }|{
        forall(c_ in carsWithConfig[config[c]]:!placeOfCar[c_].bound())
         cp.post(placeOfCar[c_]!=val);
      }
   }
}onRestart{ //LNS Relaxation
   set{int} relaxedCars();
   //.. fragment definition ..
   Solution sol = s.getSolution();
   forall(c in Cars:!relaxedCars.contains(c))
      s.post(placeOfCar[c] == placeOfCar[c].getSnapshot(sol));
}
```

FIGURE 1 – Modèle Comet pour le problème du Car Sequencing

Heuristique	rand	AI	AIP	MDI	MDIP
Taille (%)	15	10	15	15	15

TABLE 1 – taille de fragment optimale pour chaque heuristique (pourcentage du nombre de variables)

Lors de la comparaison, chaque heuristique est exécutée 30 fois par instance (6 fois pour chaque solution de départ). La comparaison des différentes stratégies utilise les informations suivantes :

- Le nombre d'améliorations
- l'amélioration moyenne
- le temps moyen d'exécution

3.3 Résultats expérimentaux

Cette section présente les résultats expérimentaux des heuristiques présentées dans ce papier sur le problème du Car Sequencing. Il faut noter que, sur ce problème, l'impact dynamique moyen et l'impact dynamique max mènent à la même sélection de fragments. En effet, l'objectif étant de maximiser le nombre de valeur différentes affectées aux variables, l'impact dynamique max est proportionnel à l'impact dynamique moyen. Seul l'impact dynamique moyen est donc présenté dans les résultats.

Les solutions de départ ont été obtenues avec une limite de 35 étapes LNS. Le tableau 1 montre les tailles de fragment obtenues avec la procédure de test. Le tableau 2 présente le nombre d'améliorations et le tableau 3, les améliorations moyennes. Dans les deux dernières tables, les meilleurs résultats sont indiqués en gras. La correspondance entre les noms des colonnes et les heuristiques est la suivante :

- rand → *fragment aléatoire*
- AI → *impact adaptatif*
- AIP → *impact adaptatif + proximité*
- MDI → *impact dynamique moyen*
- MDIP → *impact dynamique moyen + proximité*

La première conclusion que l'on peut tirer de ces résultats expérimentaux est que toutes les heuristiques présentées dans ce papier sont plus performantes que la sélection aléatoire de fragments. Cela est sans doute du au manque d'intensification inhérent à la sélection aléatoire. Les meilleures heuristiques sur ce problème sont l'impact dynamique moyen + proximité (MDIP) et l'impact dynamique moyen (MDI). Cela peut être justifié par la diversification induite par l'utilisation du MDI. En effet, cette mesure ne se basant que sur la solution courante, elle est plus variable que les autres et entraine probablement une meilleure diversification. L'ajout de la proximité à cette heuristique semble améliorer la qualité moyenne des fragments. Une inter-

instance	rand	AI	AIP	MDI	MDIP
bench_65_01	4	11	8	12	**13**
bench_65_03	11	14	10	**24**	19
bench_70_01	1	5	6	4	**12**
bench_70_03	3	4	5	**12**	10
bench_70_04	6	16	13	18	**24**
bench_75_01	3	**10**	5	6	6
bench_75_02	8	9	13	12	**15**
bench_75_03	8	13	13	6	**24**
bench_75_04	7	10	6	**24**	23
bench_80_01	8	**19**	13	6	**19**
bench_80_02	9	10	10	**23**	18
bench_80_03	6	11	11	18	**19**
bench_80_04	2	5	**14**	8	12
bench_85_01	6	6	7	**13**	11
bench_85_02	11	15	12	15	**17**
bench_85_04	8	**21**	7	18	8
bench_90_01	4	17	7	**26**	21
bench_90_02	2	10	7	**21**	17
bench_90_03	4	5	11	**19**	17
sum	111	211	178	285	**305**

TABLE 2 – Nombre d'améliorations

instance	rand	AI	AIP	MDI	MDIP
bench_65_01	0.2	0.5	0.4	1.2	**1.5**
bench_65_03	0.4	0.8	0.5	**1.6**	1.5
bench_70_01	0.0	0.3	0.3	0.3	**0.9**
bench_70_03	0.1	0.2	0.2	0.6	**0.9**
bench_70_04	0.2	0.6	0.6	1.0	**1.6**
bench_75_01	0.1	**0.5**	0.2	0.4	0.4
bench_75_02	0.4	0.5	0.6	**1.2**	1.1
bench_75_03	0.4	1.0	0.7	0.8	**2.0**
bench_75_04	0.2	0.4	0.2	**1.4**	1.2
bench_80_01	0.3	0.7	0.5	0.2	**0.9**
bench_80_02	0.3	0.4	0.4	1.0	**1.6**
bench_80_03	0.3	0.4	0.4	1.1	**2.0**
bench_80_04	0.1	0.2	**0.8**	0.3	0.4
bench_85_01	0.2	0.3	0.3	**0.7**	0.6
bench_85_02	0.4	0.9	0.6	**1.6**	1.5
bench_85_04	0.3	**1.0**	0.3	0.9	0.3
bench_90_01	0.2	1.0	0.4	**2.5**	1.7
bench_90_02	0.1	0.5	0.3	**1.4**	1.0
bench_90_03	0.3	0.3	0.6	1.2	**1.4**
global avg	0.2	0.6	0.4	1.0	**1.2**

TABLE 3 – Amélioration moyenne

Heuristic	rand	AI	AIP	MDI	MDIP
Mean time	2.0	2.0	2.3	2.3	2.1

TABLE 4 – Moyennes des temps d'exécutions en secondes

prétation possible est que la proximité permet d'obtenir des fragments plus structurés. La proximité, étant une mesure adaptative, pourrait également tempérer l'optimisme de l'impact adaptatif moyen. De ces expériences, nous pouvons également voir que l'impact adaptatif (AI) est la troisième meilleure heuristique de l'ensemble. Il semblerait qu'après 35 étapes LNS, les informations que l'impact adaptatif a collectées soient pertinentes pour choisir des fragments. Nous pouvons également remarquer que l'ajout de la proximité à cette heuristique dégrade ses performances. Les temps d'exécution moyen en secondes pour les différentes heuristiques sont fournis dans le tableau 4. Ce sont les moyennes, sur l'ensemble des exécutions, du temps total pris par les heuristiques pour calculer le fragment, geler les variables et effectuer la recherche CP. Le temps moyen d'exécution de l'heuristique aléatoire peut sembler contre-intuitif. En effet, on pourrait penser qu'en l'absence de calculs à réaliser, l'heuristique aléatoire devrait être plus rapide. Cependant, les temps reportés dépendent de la forme du voisinage car ils incluent le temps de la recherche CP. Ces temps d'exécution suggèrent que les voisinages définis par nos heuristiques sont explorés plus rapidement.

4 Conclusion

Ce travail présente diverses heuristiques adaptatives pour le choix des variables à relaxer pour la Recherche en Grand Voisinage. L'impact dynamique moyen, l'impact dynamique max et l'impact adaptatif sont définis pour mesurer l'influence d'une variable sur la valeur de la fonction objectif. La mesure de proximité est une estimation des liens que les contraintes induisent entre les couples de variables. Les heuristiques basées sur ces mesures sont regroupées en deux catégories : les heuristiques simples qui utilisent une mesure et les heuristiques combinées qui associent deux mesures pour choisir les fragments. La catégorie simple compte trois heuristiques, chacune utilisant une des notions d'impact définies. Trois heuristiques combinées ont été définies. Elles sont obtenues en combinant l'une des trois mesures d'impact avec la proximité. Toutes ces heuristiques ont été expérimentées sur une nouvelle relaxation du problème du Car Sequencing. Les résultats ont été obtenus par l'application d'une procédure de test créée pour évaluer les heuristiques indépendamment des autres paramètres du LNS. Cette procédure se base sur la détermination expérimentale, pour des solutions de départ fixées, des paramètres à éliminer. La comparaison porte sur la répétition d'une étape LNS. Les mesures du nombre de total d'améliorations et leurs qualité montrent que toutes nos heuristiques sont plus performantes qu'un choix aléatoire de fragments. Ces test révèlent que les heuristiques utilisant la proximité avec soit l'impact dynamique moyen, soit l'impact dynamique max conduisent au même choix de fragments et sont significativement meilleures que les autres sur le problème du Car Sequencing. Le temps moyen cumulé pour le calcul du fragment, le gel des variables et la recherche CP est comparables pour les heuristiques présentées et pour l'heuristique aléatoire. Il semble que la recherche CP soit plus rapide quand appliquée à des fragments sélectionnés par les heuristiques présentées ici, que quand cette recherche porte sur des fragments aléatoires. La suite de ce travail comprend l'expérimentation des heuristiques sur d'autres problèmes pour évaluer leurs performances en général. Une comparaison étendue des heuristique avec l'état de l'art LNS (LNS générique mais aussi spécifique) ainsi qu'avec d'autres approches incomplètes sera également réalisée. Ces comparaisons seront réalisées sur base de procédures de tests standard. Enfin, nous tenons à comparer la relaxation du Car Sequencing proposée avec celle décrite dans [1].

Remerciements. Nous remercions les reviewers pour leurs commentaires constructifs. Cette recherche est partiellement financée par le programme des Pôles d'Attraction Interuniversitaires (Politique scientifique belge), ainsi que par le projet FRFC 2.4504.10 du Fond National belge de la Recherche Scientifique.

Références

[1] Laurent Perron, Paul Shaw and Vincent Furnon. Propagation Guided Large Neighborhood Search. CP 2004, LNCS 3258 (2004) 468–481.

[2] Nicolas Levasseur, Patrice Boizumault and Samir Loudni. Boosting VNS with Neighborhood Heuristics for Solving Constraint Optimization Problems. LNCS 5296 (2008) 131–145

[3] Daniel Godard, Philippe Laborie and Wim Nuijten. Randomized Large Neighborhood Search for Cumulative Scheduling. AAAI 2005.

[4] Guy Desaulniers, Eric Prescott-Gagnon and Louis-Martin Rousseau. A Large Neighborhood Search Algorithm for the Vehicle Routing Problem with Time Windows. MIC 2007.

[5] Emilie Danna and Laurent Perron. Structured vs. Unstructured Large Neighborhood Search : A Case Study on Job-Shop Scheduling Problems with Earliness and Tardiness Costs. CP 2003, LNCS 2833 (2003) 817–821.

[6] Paul Shaw. Using Constraint Programming and Local Search Methods to Solve Vehicle Routing Problems CP 1998, LNCS 1520 (1998) 417–431.

[7] T. Petit, J.-C. Régin, and C. Bessière. Specific Filtering Algorithms for Over-Constrained Problems. CP 2001, LNCS 2239 (2001) 451–463.

[8] Philippe Laborie and Daniel Godard. Self-Adapting Large Neighborhood Search : Application to single-mode scheduling problems. Proceedings MISTA-07, Paris, 2007, 276-284.

Modélisation ASP pour l'analyse méthodique de réseaux géniques discrets

Nicolas Mobilia **Fabien Corblin** **Éric Fanchon** **Laurent Trilling**
UJF-Grenoble 1 / CNRS / TIMC-IMAG UMR 5525, Grenoble, F-38041, France
{Nicolas.Mobilia;Fabien.Corblin;Eric.Fanchon;Laurent.Trilling}@imag.fr

Résumé

Nous considérons des réseaux géniques décrits comme des réseaux de Thomas et en présentons une nouvelle modélisation à l'aide du paradigme de programmation logique ASP (Answer Set Programming). L'objectif est de mettre en évidence l'intérêt de cette modélisation pour la mise en œuvre d'une méthodologie rigoureuse d'analyse des réseaux géniques. D'une part il s'agit d'imposer des contraintes représentant les données biologiques et les hypothèses et d'autre part de disposer de fonctionnalités bien définies, telle que la correction d'incohérence ou l'inférence de nouvelles propriétés biologiques. Nous montrons que le paradigme ASP se révèle un bon candidat grâce à son pouvoir d'expression (en particulier pour la mise en place de défauts), à la minimalité des *answer set* et aux performances du solveur utilisé.

1 Introduction

Les connaissances biologiques sont de plus en plus étendues et l'utilisation d'outils informatiques pour construire, analyser et exploiter des modèles est devenue incontournable. Les réseaux de régulation géniques n'échappent pas à cette réalité.

L'approche traditionnelle consiste à proposer un modèle initial, où tous les paramètres sont instanciés et ensuite à confronter les comportements de ce modèle avec les résultats des expérimentations. En cas d'incompatibilité, il y a révision du modèle. Ce processus est répété jusqu'à obtenir des prédictions et des données expérimentales compatibles. Notre approche déclarative [10] se distingue clairement de cette démarche essai/erreur : elle consiste à formaliser l'ensemble des connaissances disponibles pour représenter en *intention* tous les modèles satisfaisant toutes les données. En cas d'incohérence, elle prévoit une démarche automatique de redressement. Sinon, l'exploitation de la classe des modèles cohérents est entreprise, en particulier la déduction automatique de propriétés nouvelles intéressantes pour le biologiste.

Nous nous plaçons dans le cadre d'une modélisation discrète des réseaux géniques : les réseaux de Thomas. Ces réseaux sont asynchrones et multi-valués et permettent de décrire facilement, et de manière qualitative, la dynamique des réseaux géniques.

Nous présentons ici une mise en œuvre pour notre approche basée sur le paradigme logique Answer Set Programming (ASP)[12]. L'objectif est de montrer l'intérêt de ce paradigme fondé sur une logique non monotone en illustrant en particulier 1) son pouvoir d'expression propre à une rédaction structurée et à la prise en compte d'informations partielles et d'hypothèses biologiques sous forme de défauts, 2) ses capacités d'inférences intéressantes dans la mesure où les modèles considérés sont minimaux, 3) les performances de résolution similaires à celle de solveurs SAT, sans restriction de pouvoir d'expression.

L'organisation de ce papier est la suivante : dans la partie 2, nous décrivons le formalisme de Thomas et, dans la partie 3, le paradigme ASP. Nous présentons dans la partie 4 la modélisation ASP de réseaux de Thomas à l'aide d'un réseau simple. Dans la partie 5, nous montrons comment modéliser en ASP des hypothèses et des observations biologiques. La partie 6 est consacrée à la méthodologie de construction de modèles et à la mise en œuvre des fonctionnalités nécessaires. La partie 7 présente les résultats obtenus sur le plan biologique et sur celui des performances pour des modélisations de réseaux géniques réels.

2 Réseaux de Thomas

Nous présentons ici une description succincte des réseaux de Thomas [16], suffisante pour notre propos.

On trouvera dans [9][10] une description plus détaillée.

Ce type de réseau fournit une formalisation discrète des réseaux géniques très acceptée, dans la mesure où l'on peut dire qu'elle représente une abstraction des systèmes d'équations différentielles non-linéaires à base de sigmoïdes, décrivant l'évolution des concentrations de protéines produites à partir des gènes [16].

La structure des réseaux de régulation géniques est habituellement représentée par un graphe dirigé dit *d'interaction*, dans lequel les nœuds représentent les gènes et les arcs représentent les interactions entre gènes. Soient deux gènes i et j, connectés par un arc allant du gène j au gène i. Cet arc indique que le gène j peut potentiellement modifier le taux d'expression du gène i. Il est étiqueté par une valeur entière identifiant le seuil discret de j auquel une modification du taux d'expression du gène cible i se produit. Une interaction est médiée par une protéine régulatrice, chaque gène produisant une seule protéine.

La dynamique du réseau de gènes est décrite par un graphe dirigé dit de *transition*. Dans ce graphe, chaque nœud est un état du système et chaque arc est un passage possible d'un état vers un autre. Un état du système est décrit par une liste $S = [x_1, \ldots, x_n]$ contenant les concentrations des protéines associées aux n gènes du réseau. Les concentrations sont représentées par des valeurs discrètes correspondant à un intervalle entre deux seuils consécutifs du gène. Par exemple, si la variable x_i vaut 0, cela signifie que la concentration de la protéine codée par le gène i est inférieure au plus petit seuil de i.

Pour un état donné du système, la concentration vers laquelle tend chaque protéine est donnée par l'*état focal* associé à l'état courant du système. Nous notons X_i la *valeur focale* du gène i. L'état focal d'un état du système est donné par une liste $X = [X_1, \ldots, X_n]$ et représente l'état vers lequel *tend* le système dans l'état courant (ce n'est pas forcément le successeur de l'état courant).

Plus précisément, la valeur focale d'un gène i est déterminée par la concentration des protéine j tel qu'il existe un arc $j \rightarrow i$ dans le graphe d'interaction. Pour chaque gène, nous définissons un ensemble de contextes cellulaires : un contexte cellulaire d'un gène i est un ensemble d'états du système dans lesquels le taux de production de i est constant. Soient $j_1, ..., j_p$ les gènes agissant sur le gène i avec les seuils $\theta_{j_1}, ..., \theta_{j_p}$. L'ensemble des contextes cellulaires du gène i constitue une partition de l'espace des concentrations où chaque contexte cellulaire est défini par un ensemble d'inégalités : $x_{j_1} \mathsf{op_1} \theta_{j_1}, \ldots, x_{j_p} \mathsf{op_p} \theta_{j_p}$ où $\mathsf{op_1} \ldots \mathsf{op_p} \in \{<\ ,\ \geq\}$, x_{j_1} est la concentration du gène j_1 et θ_{j_1} est le seuil de j_1 auquel se produit une variation du taux d'expression de i. Dans l'exemple montré par la figure 1, au gène b correspond quatre contextes cellulaires :

- un premier tel que $x_a < \theta_a^1$ et $x_b < \theta_b^2$,
- un deuxième tel que $x_a < \theta_a^1$ et $x_b \geq \theta_b^2$,
- un troisième tel que $x_a \geq \theta_a^1$ et $x_b < \theta_b^2$,
- un quatrième tel que $x_a \geq \theta_a^1$ et $x_b \geq \theta_b^2$.

Au contexte cellulaire c_i d'un gène i est associé un paramètre cinétique correspondant à la focale de i. Nous désignons ce paramètre par $K_i^{dessus(c_i)}$ où $dessus(c_i)$ est l'ensemble des noms des gènes j_r influençant i et tels que x_{j_r}, dans le contexte cellulaire c_i, soit supérieur ou égal au seuil θ_{j_r} étiquetant l'arc $j_r \rightarrow i$. Dans notre exemple, les paramètres cinétiques correspondant aux contextes cellulaires présentés plus haut sont, dans l'ordre de présentation des contextes : K_b, K_b^b, K_b^a et K_b^{ab}.

Soit $contexte(S, c'_i)$ tel que pour, tout contexte cellulaire c'_i du gène i : $contexte(S, c'_i) = 1$ si l'état du système S appartient au contexte cellulaire c'_i, 0 sinon. La valeur focale de i est définie comme :

$$X_i = \sum_{c'_i} K_i^{dessus(c'_i)} * contexte\,(S, c'_i)$$

Les équations focales des deux gènes de l'exemple 1(a) sont montrées en 1(b).

Le formalisme de Thomas est un formalisme non déterministe. Si deux gènes tendent à changer de concentration, deux successeurs sont possibles : un pour lequel le premier gène change de valeur, et un pour lequel le deuxième gène change de valeur. En effet, dans la réalité, il est très peu probable que deux gènes franchissent leur seuil exactement au même moment. Pour un état donné S, l'ensemble des successeurs de S est déterminé en comparant la valeur d'expression de chaque gène avec sa valeur focale : $S' = [x'_1, \ldots, x'_n]$ est un successeur de S si :

- Soit il existe i tel que $X_i \neq x_i$, dans ce cas, pour tout j tel que $j \neq i$, $x'_j = x_j$ et (1) $x'_i = x_i + 1$ si $X_i > x_i$ ou (2) $x'_i = x_i - 1$ si $X_i < x_i$.
- Soit pour tout j, $X_j = x_j$, alors, pour tout j, $x'_j = x_j$: S est dit *stationnaire*.

3 Answer set programming

ASP [1][12] est un langage apparu vers la fin des années 1990 comme un paradigme déclaratif. Il est basé sur une logique non monotone définie à l'aide de la notion de modèles *stables*. Nous en faisons une présentation succincte ici.

Un programme logique ASP est un ensemble fini de règles de la forme :
$a_0 \leftarrow a_1, \ldots, a_m, not\ a_{m+1}, \ldots, not\ a_n$
où $0 \leq m \leq n$ et $\forall i \mid 0 \leq i \leq n,\ a_i$ est un atome. Pour

une règle r, $tete(r) = a_0$ est la tête de la règle, et $corps(r) = \{a_1, \ldots, a_m, not\ a_{m+1}, \ldots, not\ a_n\}$ est le corps de cette règle (i.e. la partie droite de la règle). Si $tete(r)$ est vide, r est une *contrainte d'intégrité*. Si $corps(r)$ est vide, r est un *fait*.

Soit A l'ensemble des atomes, $corps^+(r) = \{a \in A \mid a \in corps(r)\}$ et $corps^-(r) = \{a \in A \mid not\ a \in corps(r)\}$. Un ensemble $X \subseteq A$ est un *answer set* ou *modèle stable* d'un programme P si X est le modèle minimal du *réduit* $P^X = \{tete(r) \leftarrow corps^+(r) \mid r \in P, corps^-(r) \cap X = \varnothing\}$. [1]

Exemple 1 : soit le programme E suivant :

```
a :- not b,c.
b :- not a.
c.
```

Soit $X = \{a, c\}$. Nous obtenons le réduit $E^X = \{c,\ a \leftarrow c\}$ dont le modèle minimal est $\{a, c\}$. X est un modèle stable de E.
Soit $X' = \{a, b, c\}$. Nous obtenons le réduit $E^{X'} = \{c\}$ dont le modèle minimal est $\{c\}$. X' n'est pas un modèle stable de E.

Exemple 2 : le programme E' suivant :

```
a :- not b.
b :- not a.
```

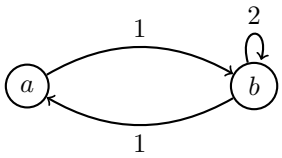
a deux modèles stables $\{a\}$ et $\{b\}$. Si nous ajoutons la contrainte d'intégrité `:- a.`, nous éliminons le modèle $\{a\}$. Si nous ajoutons la contrainte d'intégrité `:- not a.`, nous éliminons le modèle $\{b\}$ car il ne contient pas a.

Ce formalisme permet également d'exprimer des contraintes de cardinalité sur le nombre de littéraux vrais. Si nous imposons la contrainte `u{l_1,...,l_n}v.`, nous n'obtenons que des modèles où le nombre de littéraux `l_i` vrais est compris entre u et v.

D'un point de vue opérationnel, le logiciel que nous utilisons [12] procède en deux étapes pour calculer les answer sets d'un programme P. Dans la première, un « grounder » qui substitue aux variables du programme des termes sans variables libres, produit un programme propositionnel $\mathbf{P}$ correspondant à P. Dans la seconde, un solveur calcule les answer sets de $\mathbf{P}$.

Le solveur que nous utilisons [12] propose l'opérateur `#maximize` (resp. `#minimize`) maximisant (resp. minimisant) le nombre de formules atomiques vraies dans un ensemble. Par exemple, si nous imposons `#maximize{f_1,...,f_n}`, nous obtenons un modèle contenant le plus grand nombre possible de formules atomiques `f_i` vraies. S'il existe plusieurs modèles, un seul d'entre eux est fourni.

1. Un ensemble de clauses de Horn ne contenant pas de littéral négatif possède toujours un modèle minimal (tel que si on lui soustrait un ou plusieurs atomes, on ne peut obtenir un modèle) et ce modèle est unique. Un réduit P^X correspond à un tel ensemble.



(a)

$$
\begin{aligned}
X_a = \quad & K_a * s^-(x_b, \theta_b^1) + \\
& K_a^b * s^+(x_b, \theta_b^1) \\
X_b = \quad & K_b * s^-(x_a, \theta_a^1)\ s^-(x_b, \theta_b^2) + \\
& K_b^a * s^+(x_a, \theta_a^1)\ s^-(x_b, \theta_b^2) + \\
& K_b^b * s^-(x_a, \theta_a^1)\ s^+(x_b, \theta_b^2) + \\
& K_b^{ab} * s^+(x_a, \theta_a^1)\ s^+(x_b, \theta_b^2)
\end{aligned}
$$

(b)

FIGURE 1 – (a) Exemple de graphe d'interaction d'un réseau de régulation génique. Le gène a active l'expression du gène b, tandis que le gène b inhibe l'expression du gène a. De plus, le gène b s'auto-active. L'étiquette « 1 » sur l'arc $b \to a$ indique que l'expression de a change lorsque la concentration de b traverse le premier seuil de b. (b) Les équations focales de cet exemple définissant l'état focal $[X_a, X_b]$ du système, lorsque celui-ci est dans un état $[x_a, x_b]$. $s^+(x, \theta)$ vaut 1 si $x \geq \theta$ et 0 sinon, et $s^-(x, \theta) = 1 - s^+(x, \theta)$

4 Modélisation en ASP d'un réseau de Thomas

Pour cette spécification, nous procédons en introduisant d'abord des règles donnant le domaine de valeur des atomes et, ensuite, des règles d'intégrité limitant le nombre de modèles possibles.

Par convention de nommage, les notations `predicat(X,Y)` et `predicat/2` sont équivalentes.

4.1 Graphe d'interaction

Les réseaux de régulation sont souvent représentés par un graphe d'interaction : la figure 1 en donne un exemple simple. Pour modéliser ce graphe, nous utilisons les prédicats `node(N)` vrai ssi `N` est un nœud du graphe d'interaction, `edge(N1,N2,No)` vrai ssi il existe un arc n° `No` allant du nœud `N1` au nœud `N2` (le paramètre `No` permet de modéliser des multi-arcs) et `threshold(N1,N2,No,T)` vrai ssi `T` est l'unique seuil, strictement positif, de l'arc n° `No` allant de `N1` à `N2`.

Le graphe d'interaction de l'exemple (Fig. 1) est modélisé par les formules atomiques suivantes : `node(a). node(b). edge(a,b,1). edge(b,a,1). edge(b,b,1). threshold(a,b,1,1). threshold(b,a,1,1). threshold(b,b,1,2).`

Si la valeur d'un seuil n'est pas imposée, l'ensemble des valeurs possibles pour ce seuil est considéré, chaque

valeur étant attribuée à au moins un modèle différent. Cette modélisation permet d'imposer des contraintes sur les seuils.

4.2 Dynamique du système

Un moyen de description de l'évolution des états du système est nécessaire pour saisir la dynamique du réseau. De plus, si nous possédons des données biologiques sur la dynamique, il est nécessaire de les représenter pour les inclure dans notre modèle.

L'évolution du système est traduite par des chemins de longueur finie. Pour cela, nous introduisons les prédicats `path(P)` vrai ssi `P` est un chemin, `length(L,P)` vrai ssi le chemin `P` est de longueur `L` et `step(I,P)` vrai ssi le chemin `P` possède une étape n^o `I`. À chaque étape d'un chemin, le système est dans un état donné. La description d'un état se fait grâce au prédicat `species(N,V,I,P)` vrai ssi à l'étape `I` du chemin `P`, la concentration du gène `N` est `V`. Pour chaque gène, l'intervalle de concentrations possibles est défini grâce au prédicat `val(N,V)` vrai ssi `V` est une valeur possible de concentration pour `N`.

Définir la dynamique du système nécessite de spécifier l'ensemble des successeurs possibles d'un état. Comme décrit dans la partie 2, dans le formalisme utilisé, la concentration d'une protéine associée à un gène changeant de valeur varie selon la tendance donnée par la focale de ce gène. Pour cela nous introduisons le prédicat `focal(N,F,I,P)` vrai ssi, à l'étape `I` du chemin `P`, la valeur focale du gène `N` est `F`.

La valeur focale d'un gène est définie grâce aux paramètres cinétiques associés aux contextes cellulaires de ce gène. Pour modéliser ces paramètres cinétiques, nous distinguons l'identifiant d'un paramètre et sa valeur en introduisant les prédicats `param(N,Ik)` vrai ssi `Ik` est l'identifiant d'un paramètre cinétique du gène `N` et `kparam(K,Ik)` vrai ssi le paramètre d'identifiant `Ik` vaut `K`. Pour un gène donné, tous les paramètres cinétiques sont décrits par un identifiant pour exprimer des contraintes relatives aux interactions (voir § 5.2). Nous imposons (1) l'existence d'une et une seule valeur par paramètre cinétique et (2) que cette valeur soit comprise entre 0 et la valeur du plus grand seuil du gène en question. Pour imposer (1), nous utilisons la contrainte de cardinalité suivante :

```
1{kparam(K,Ik):val(N,K)}1 :-
                param(N,Ik), node(N).
```

Cette contrainte impose que, pour un identifiant `Ik`, parmi toutes les valeurs possibles de `K`, il y ait une et une seule formule atomique `kparam(K,Ik)` qui soit vraie. Pour imposer (2), nous utilisons la contrainte d'intégrité suivante :

```
:- kparam(K,Ik), param(N,Ik), val(N,K),
    not threshold(N,N2,Ne,K):edge(N,N2,Ne),
    node(N), K!=0.
```

Dans notre exemple, le gène `a` possède deux contextes cellulaires, un pour lequel `b` est en dessous de son premier seuil, et un pour lequel `b` est au dessus de ce seuil. Les identifiants des paramètres cinétiques correspondant à chaque contexte sont définis par les formules atomiques suivantes : `param(a,ka_).` et `param(a,ka_b).` Pour le gène `b`, la contrainte (2) impose que les valeurs possibles vers lesquelles tend ce gène sont comprises entre 0 et 2 puisque `b` possède deux seuils.

Grâce aux prédicats `param/2` et `kparam/2`, nous pouvons définir les règles calculant la focale d'un gène selon son contexte cellulaire. Dans notre exemple, la focale du gène `a` dans le contexte cellulaire où `b` est au dessus de son seuil est déterminée par la règle :

```
focal(a,Ka_b,I,P) :- species(b,Vb,I,P),
  above(b,a,A,Vb), step(I;I+1,P),
  param(a,ka_b), kparam(Ka_b,ka_b).
```

Où le prédicat `above(A,B,No,V)` est vrai ssi `V` est supérieur ou égal au seuil de l'arc `No` allant de `A` à `B`.

Le formalisme de Thomas étant non-déterministe, la concentration d'au plus une protéine change de valeur. De plus, si la focale d'un gène est différente de la concentration de la protéine associée, l'état du système n'est pas un état stationnaire. Dans notre modélisation, nous assurons simplement ces propriétés en utilisant le prédicat `diff(N,I,P)` vrai ssi la protéine `N` est la protéine dont la concentration change selon la tendance de sa focale à l'étape `I` du chemin `P`. La contrainte de cardinalité suivante :

```
0{diff(N,I,P):node(N)}1 :- step(I;I+1,P).
```

impose qu'au plus une seule concentration change entre un état et son successeur. En introduisant le prédicat `foceg(N,I,P)` vrai ssi la concentration de la protéine `N` est égale à sa valeur focale à l'étape `I` du chemin `P`, la contrainte d'intégrité :

```
:- 1{not foceg(N,I,P):node(N)},
    0{diff(N,I,P):node(N)}0, step(I;I+1,P).
```

permet d'assurer qu'en cas d'égalité d'un état et de son état focal, l'état en question est stationnaire.

Pour l'exemple présenté dans la figure 1, nous obtenons huit instanciations différentes des paramètres cinétiques. Ces huit instanciations correspondent à six graphes de transitions différents, nommés G_1 à G_6, présentés dans la figure 2.

5 Modélisation en ASP d'observations et d'hypothèses biologiques

Une fois le modèle de Thomas représenté en ASP, nous devons modéliser les données biologiques.

5.1 Chemins

Les données sur le comportement du réseau peuvent, en général, s'exprimer par des propriétés sur des chemins. Nous pouvons imposer des contraintes sur les concentrations des protéines dans les étapes d'un chemin de deux manières différentes :

- Nous pouvons imposer des contraintes « instanciées » : par exemple, imposer que la concentration d'une protéine pour une étape donnée d'un chemin ait une valeur donnée. Pour cela, nous utilisons une contrainte d'intégrité sur un atome `species(N,V,I,P)`.
- Nous pouvons également imposer des contraintes « non instanciées », comme dans le cas ci-dessus mais avec, par exemple, `V` restreint à `V<2` ou encore, comme l'existence d'un état stationnaire.

Pour exprimer l'existence d'un état stationnaire, nous imposons l'existence d'un chemin de longueur 2 telle que l'état du système à l'étape 1 et à l'étape 2 soit le même. Pour cela, nous définissons le prédicat `statpath(P)` vrai ssi `P` est de longueur 2 et est stationnaire. Pour implémenter ce prédicat, nous définissons le prédicat `succeg(N,P)` vrai ssi `P` est un chemin de longueur 2 et la concentration du gène `N` est la même à l'étape 1 et à l'étape 2. Grâce à ce prédicat, nous définissons `statpath/1` avec à la contrainte d'intégrité suivante :

```
:- statpath(P), 1{not succeg(N,P):node(N)}.
```

qui impose que pour un chemin `chemin`, il est impossible que `statpath(chemin)` soit vrai et qu'il existe un gène dont la concentration ne soit pas la même dans les deux étapes de `chemin`.

Nous pouvons remarquer que nous n'avons, ici, imposé aucune valeur de concentration des protéines, mais nous avons pu imposer que cette valeur soit la même dans les deux étapes du chemin grâce à la contrainte d'intégrité définissant `statpath/1`.

Ces contraintes traduisant les observations biologiques permettent de rejeter toutes les instanciations ne vérifiant pas ces observations.

Plusieurs logiques temporelles exprimant des propriétés de chemins ont été conçues, en particulier CTL (Computational Tree Logic)[7] proposée dans le cadre des réseaux biologiques [2][6]. Intuitivement, elle permet de formuler des propriétés du type : « il existe un (resp. tout) chemin possédant (resp. possède) telle caractéristique » où une caractéristique est une propriété de tous les états de chemin ou d'un au moins. Par exemple, dans le cadre de l'exemple simple (Fig. 1), la formule $(a = 0 \wedge b = 0) \Rightarrow EF(a = 0 \wedge b = 2)$ signifie qu'il doit exister (EF : Exist Future) un chemin débutant dans un état où $a = 0$ et $b = 0$ et atteignant un état où $a = 0$ et $b = 2$. Cette propriété est facilement imposée en ASP en introduisant un chemin `p` de taille maximum 5 sous la forme suivante :

```
path(p). length(5,p).
 :- not species(a,0,1,p).
 :- not species(b,0,1,p).
exist_path :- species(a,0,I,p),
        species(b,2,I,p), step(I,p).
```

et en utilisant la contrainte d'intégrité :

```
 :- not exist_path.
```

Seuls les modèles G_4 et G_6 satisfont cette formule.

On note que le nombre d'atomes de la forme `species(N,V,I,P)` reste linéaire en fonction de la taille du chemin `P`.

Vérifier une propriété CTL universelle est théoriquement possible, mais est peu intéressant, car, du fait de l'abstraction utilisée [16], si nous imposons une propriété de ce type, certains modèles peuvent être rejetés à tort.

5.2 Signes des interactions

Dans les papiers utilisant le formalisme de Thomas, tous les arcs du graphe d'interaction sont étiquetés par un signe « + » ou « − ». Si un arc $j \to i$ est étiqueté par un signe « + » (resp. un signe « − »), cela indique intuitivement que le gène j active (resp. inhibe) le gène i au moins dans un contexte cellulaire. Néanmoins, le sens exact des termes activation et inhibition n'est pas défini précisément, et plus particulièrement dans le cas où d'autres gènes régulent aussi l'expression de i : est-ce que l'activation de i par j empêche l'inhibition de i par j ? Est-ce que, quelles que soient les interactions que i subit, j active toujours i ? Nous modélisons formellement ces questions en exprimant deux types de contraintes :

Le premier type de propriété que nous considérons traduit l'observation biologique d'une activation (resp. inhibition) de i par j. Nous modélisons cette propriété de la façon suivante : nous imposons qu'il existe deux contextes cellulaires $c1_i$ et $c2_i$ de i tels que :

- dans $c1_i$, la concentration de j est inférieure au seuil θ_j,
- $c2_i$ est identique à $c1_i$ excepté pour j, dont la concentration est supérieure à θ_j,

alors, les paramètres cinétiques associés à ces contextes cellulaires sont tels que, si j active (resp. inhibe) i, la tendance de i dans $c1_i$ est strictement inférieure (resp. strictement supérieure) à la tendance de i dans $c2_i$. De plus, si un gène i s'auto-active (resp. s'auto-inhibe) avec un seuil θ_i, sa valeur focale, dans le contexte cellulaire où i est supérieur (resp. inférieur) à θ_i doit être supérieure (resp. inférieure) à ce seuil, sinon, le système arrive toujours dans un état où la concentration de i est inférieure (resp. supérieure) à θ_i et l'effet de l'auto-activation n'est pas observable.

Pour exprimer cette propriété, nous imposons des contraintes dites *contraintes d'observabilité*. Pour notre exemple, pour le gène b, les contraintes d'observabilité sont :
$(K_b < K_b^a) \vee (K_b^b < K_b^{ab})$ et
$((K_b < K_b^b) \wedge (K_b^b \geq 2)) \vee ((K_b^a < K_b^{ab}) \wedge (K_b^{ab} \geq 2))$.

Le deuxième type de propriété que nous modélisons exprime que pour tout couple de contextes cellulaires $c1_i$, $c2_i$ de i, satisfaisant aux mêmes contraintes que précédemment, alors, dans le cas d'une activation (resp. inhibition), la tendance de i dans le contexte cellulaire $c1_i$ n'est pas supérieure (resp. inférieure) à la tendance de i dans le contexte cellulaire $c2_i$. Intuitivement, cela signifie qu'il est impossible d'avoir deux paramètres cinétiques pour $c1_i$ et $c2_i$ qui vérifieraient la contrainte d'observabilité imposée par j inhibe (resp. active) i. Cela signifie également que, dans le cas ou deux gènes régulent de la même manière un troisième gène (positivement ou négativement), la régulation combinée de ces deux gènes est au moins aussi forte que la régulation de chaque gène individuellement.

Nous exprimons cette propriété par des contraintes dites *contraintes d'additivité*. Dans notre exemple, les contraintes d'additivité portant sur b sont : $K_b \leq K_b^b$, $K_b^a \leq K_b^{ab}$, $K_b \leq K_b^a$ et $K_b^b \leq K_b^{ab}$.

Les contraintes d'additivité représente un cas typique de règles « générales » en ce qu'elles s'appliquent sauf exception. Par exemple, comme il a été dit, il est assez exceptionnel que la régulation combinée de deux gènes sur un troisième ne soit pas au moins aussi forte que la régulation de chaque gène individuellement. Cependant, biologiquement, le cas est possible : par exemple, si les sites de fixation sur l'ADN des protéines correspondantes se chevauchent. La technologie ASP se prête très bien à la formalisation de ce type de connaissance dans la mesure où elle est basée sur une logique non monotone permettant l'expression de défauts [4], plus précisément ici de défauts *normaux* exprimant que « telle contrainte d'additivité est vraie jusqu'à preuve du contraire ». Un tel défaut pour la contrainte d'additivité $K_b \leq K_b^a$ se formule ainsi :
`addit(b,kb_,kb_a) :- not -addit(b,kb_,kb_a).`
où : `addit(b,kb_,kb_a)` implique $K_b \leq K_b^a$,
`-addit(b,kb_,kb_a)` et `addit(b,kb_,kb_a)`
sont exclusifs (i.e.
`:- -addit(b,kb_,kb_a), addit(b,kb_,kb_a).`),
`-addit(b,kb_,kb_a)` est impliqué par $K_b > K_b^a$. En d'autres termes, si $K_b > K_b^a$ n'est pas démontrable alors $K_b \leq K_b^a$ est vrai.

5.3 Mutants

Pour étudier des réseaux géniques, les biologistes sont amenés couramment à supprimer un gène ou à faire en sorte qu'il soit sur-exprimé. Les nouveaux réseaux obtenus sont dits *mutants* par opposition au réseau initial dit *sauvage*. Dans cette partie, nous entendons par « modèle » un réseau qui peut être sauvage ou mutant. Le problème consiste à définir un modèle mutant d'un réseau sauvage et à exprimer les propriétés des différents modèles.

La démarche procède par extension. Elle consiste à introduire les prédicats `model(M)` vrai ssi `M` est un modèle, `mutant(N,M,V)` vrai ssi, dans le modèle `M`, `N` est un gène mutant dont la valeur d'expression est forcée à `V` et `mutant(N,M)` vrai ssi `N` est un gène mutant du modèle `M`.

Pour différencier les chemins du modèle sauvage des chemins d'un modèle mutant, nous étendons le prédicat `path/1` en `path(P,M)` vrai ssi `P` est un chemin dans le modèle `M`.

Il est proposé dans [9] de définir de nouveaux paramètres cinétiques pour chaque mutant et d'imposer l'égalité des paramètres entre le modèle sauvage et le modèle mutant pour les gènes qui ne sont pas mutants. Cette approche est simple, mais présente l'inconvénient de devoir créer beaucoup de paramètres.

Le fait d'utiliser une logique non-monotone permet d'éviter de définir des paramètres cinétiques supplémentaires et de n'écrire que des équations focales spécifiques pour chaque mutant : en effet, il suffit d'utiliser les équations focales du modèle sauvage, sauf pour les gènes présentant une mutation. Nous implémentons ceci en rajoutant en partie droite des règles définissant les focales, le littéral `not mutant(N,M)`. Cela indique que nous utilisons ces règles, sauf dans les cas où le gène `N` est un mutant dans le modèle `M`. Nous définissons alors, la valeur focale d'un gène mutant comme étant sa valeur de mutation, de plus, nous imposons que, dans l'état initial des chemins, la concentration des gènes mutants soit celle imposée par la mutation.

6 Traitement d'incohérence et déduction de propriétés

La stratégie utilisée dans notre approche prévoit de définir un ensemble initial de contraintes traduisant les équations focales, les contraintes d'additivité et d'observabilité, les données biologiques, et des hypothèses sur le fonctionnement du réseau.

À partir de cet ensemble initial, soit nous obtenons au moins un modèle, soit nous n'en obtenons aucun. Dans ce dernier cas, cela signifie qu'il y a une incohérence entre le graphe d'interaction, les données biologiques et les hypothèses. Nous séparons les contraintes en deux catégories : celles qui sont soutenues par des données biologiques et donc non relâchables, et celle qui ne le sont pas (hypothèses) ou qui sont

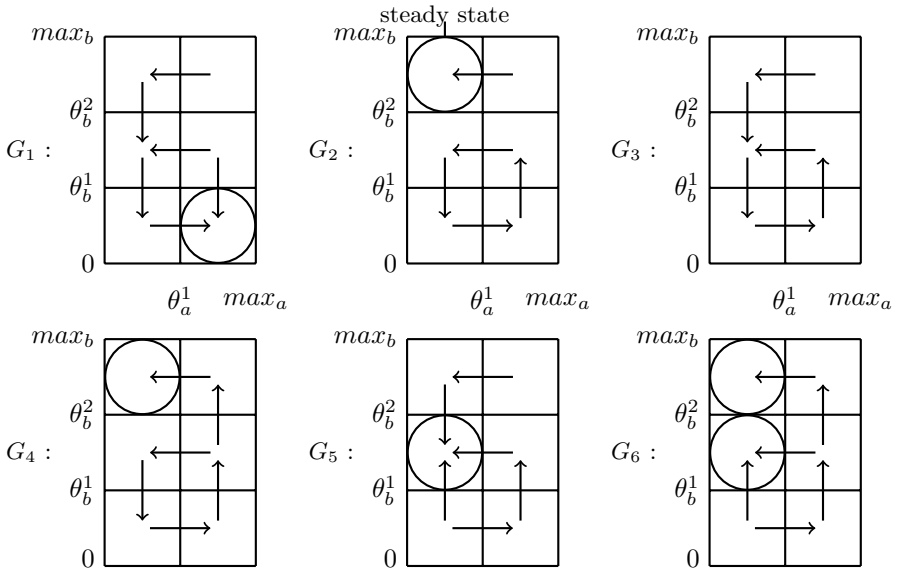


FIGURE 2 – Graphes de transition G_1, ..., G_6 satisfaisants l'ensemble des contraintes d'observabilité et d'additivité du réseau présenté par Fig. 1. Chaque case correspond à un état du système. Les états stationnaires sont indiqués par un rond. Les flèches représentent les transitions possibles. Chaque graphe correspond à une ou deux instanciations des paramètres cinétiques. Par exemple, le graphe G_4 correspond à l'instanciation suivante : $K_a = 1, K_a^b = 0, K_b = 0, K_b^b = 2, K_b^a = 2, K_b^{ab} = 2$.

moins fiables. Le but est alors de trouver le plus petit ensemble de contraintes à relâcher dans l'ensemble des contraintes relâchables pour que l'ensemble de contraintes restantes soit cohérent.

Pour cela, nous devons procéder en deux étapes :

- premièrement, la détermination du nombre minimum de contraintes relâchables. Considérons que nc contraintes soient relâchables. Soit `cv_`i, i∈[1..nc] vrai ssi la i^e contrainte relâchable est vrai. La commande `#maximize{cv_1,...,cv_n}` permet d'obtenir le nombre N maximum de contraintes pouvant être respectées.
- deuxièmement, comme nous disposons du nombre N, nous posons la contrainte de cardinalité : N{`cv_1,...,cv_n`}N.

S'il existe différents ensembles de contraintes à relâcher menant à la consistance, nous procédons à une analyse manuelle de ces ensembles, afin d'évaluer leur pertinence biologique et considérons chacun de ces ensembles, sauf ceux dont la relaxation n'aurait pas de sens biologique. Il parait néanmoins difficile de fournir plus d'une dixaine d'ensembles de contraintes à relâcher à un biologiste sans fournir des éléments d'analyse supplémentaires

Il faut noter que le redressement d'incohérence est toujours gênant, car il nécessite un cadre para-logique. Ici, il peut être simplement évité en représentant les hypothèses sous la forme de défauts (voir §5.2) : par exemple, si effectivement $K_b > K_b^a$ est prouvé, un modèle différent de ceux correspondant aux graphes de la Fig. 2 est proposé. L'intérêt de rester dans le même cadre logique est clair : au lieu d'un résultat d'incohérence, nous obtenons une indication sur les hypothèses contredites et sur les conséquences de leur remise en cause.

À partir d'un ensemble de contraintes cohérentes, notre approche propose plusieurs fonctionnalités, telle que la déduction de propriétés vraies dans tous les modèles ou la recherche du nombre minimal de seuils.

La déduction automatique de propriétés est mise en œuvre facilement grâce à l'option `--cautious` proposée par [12]. Cette option renvoie l'ensemble des formules atomiques vraies dans tous les modèles. En spécifiant un langage décrivant les propriétés que nous recherchons, nous obtenons l'ensemble des propriétés vraies pour tous les modèles.

Un tel langage peut être construit à l'aide de clauses [5]. Inférer des propriétés revient alors à inférer des clauses. Il est toutefois critique de choisir pertinemment l'ensemble de formules atomiques. En [10], les auteurs proposent deux langages permettant d'exprimer des disjonctions d'inégalités entre paramètres cinétiques, comme $K_b < K_b^a \vee \neg(K_b < K_b^{ab})$.

Par exemple, supposons que nous voulions obtenir

toutes les clauses de taille 2 portant sur des atomes qui sont des inégalités strictes sur des paramètres cinétiques d'un même gène. Il suffit pour cela de définir le prédicat `clauses2(In,Ik1,Ik2,Inp,Ik1p,Ik2p)` où `In` et `Inp` représentent des opérateurs d'inégalité (non strictes) et où `Ik1` et `Ik2` (resp. `Ik1p` et `Ik2p`) sont des identificateurs de paramètres cinétiques d'un même gène, vrai si une clause formée à partir des deux atomes correspondant est démontrée.

Il faut noter la spécificité et l'intérêt de la technologie ASP en cette matière d'inférence de propriétés communes à un ensemble de modèles. On sait que les modèles obtenus sont minimaux en ce sens que soustraire un ou plusieurs atomes à un modèle ne peut donner un modèle. Par exemple, `a :- not b.` `b :- not a.` n'a que deux modèles : {`a`} et {`b`}. Le modèle {`a`, `b`} n'est pas minimal. Il en ressort que le nombre de propriétés déductibles est égal ou supérieur à celui atteint en logique classique. Ainsi l'exemple précédent permet de conclure à l'exclusion de `a et b` (en l'absence d'autre information sur `a` et `b`), ce qui ne serait pas le cas si le seul axiome était l'union classique de `a` et `b`.

La recherche du nombre minimum de seuils est mise en œuvre simplement en utilisant l'opérateur `minimize` proposé par [12]. En effet, en définissant le prédicat `threshold_max(N,T)` vrai ssi `T` est le nombre de seuils du gène `N`, nous obtenons le minimum du nombre total de seuils en utilisant cet opérateur ainsi :

```
#minimize
    [threshold_max(N,T):val(N,T):node(N)=T].
```

À chaque atome `threshold_max(N,T)` vrai, est associé le poids `T` et il s'agit de minimiser la somme de ces poids.

7 Applications et Résultats

Nous présentons dans cette section deux applications illustrant les fonctionnalités proposées par notre approche.

7.1 Stress nutritionnel de *E. coli*

Escherichia coli est une bactérie dont la population croit exponentiellement en présence de nourriture. Dans le cas d'un stress nutritionnel, la bactérie stoppe cette croissance et entre dans une phase dite stationnaire où elle ne se reproduit plus. La réponse à ce stress est réversible : si la source nutritionnelle est à nouveau disponible, la bactérie retourne en phase exponentielle. Cette adaptation à l'environnement est due à une modification de la concentration des gènes de cette bactérie en fonction de la présence ou de l'absence de nourriture.

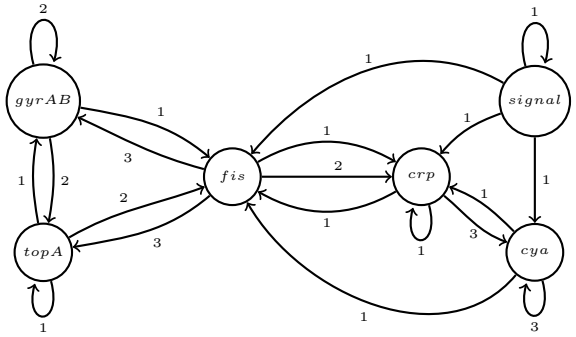


FIGURE 3 – Graphe d'interaction du réseau de régulation génique d'E. coli.

L'article [14] présente un ensemble d'observations biologiques et propose un réseau de gènes issu de ces observations. Le graphe d'interaction de ce réseau est présenté dans la figure 3. Une étude avec une approche déclarative de ce réseau est présentée en [10]. Nous avons repris cette étude avec notre mise en œuvre ASP. Nous illustrons, ici, la relaxation de contraintes due à une incohérence dans les données.

Parmi les contraintes portant sur ce réseau, on trouve, en particulier :

- l'existence de deux états stationnaires partiellement connus correspondant à l'état stressé et non-stressé de la bactérie.
- une contrainte portant sur les concentrations des gènes dans les états stationnaires
- l'existence de deux chemins, chacun menant d'un état stationnaire à l'autre

Nous modélisons ce réseau de régulation dans notre formalisme et nous constatons qu'il n'existe pas d'instanciation des paramètres rendant ce système cohérent. Il est alors intéressant de rechercher le plus petit ensemble de contraintes à relâcher pour obtenir un modèle. Les contraintes d'additivité n'étant pas soutenues par des données expérimentales, nous recherchons donc à relâcher des contraintes d'additivité.

Il apparaît que la relaxation d'une contrainte parmi $\{C_{gyrAB1}, C_{topA1}\}$ est suffisante. $C_{gyrAB} = K^{fis}_{gyrAB} \leq K_{gyrAB}$ est une contrainte d'additivité portant sur le gène *gyrAB*. $C_{topA} = K_{topA} \leq K^{fis}_{topA}$ est une contrainte d'additivité portant sur le gène *topA*. Une analyse biologique montre que cette dernière, à la différence de C_{gyrAB}, peut être tout à fait relâchée dans l'état des connaissances.

L'étude menée en [10] est basée sur une mise en œuvre réalisée à l'aide de l'outil GNBox [8] qui assure une coopération entre un solveur de contrainte

Prolog et un solveur SAT. La modélisation d'un réseau de Thomas est traduite sous forme clausale ainsi que certaines contraintes biologiques. D'un point de vue performance, on note en [10] qu'un temps très significatif (25 minutes pour la traduction en clauses et la résolution) est nécessaire pour déterminer les contraintes relâchables citées ci-dessus, dans un cas où toutes les contraintes d'additivité sur le gène *crp* sont supprimées. On trouve en [9] un temps meilleur (sept secondes ; sur un laptop, 2.4GHz, 2Go de RAM) pour la même tâche avec l'outil GNBox amélioré où toutes les contraintes sont traduites sous forme clausale. Avec notre implémentation, cette tâche est accomplie en moins de quatre secondes (avec un Core 2 Duo 3GHz, 4Go de RAM).

Il en ressort tout l'intérêt de disposer d'un langage (ASP), d'une part pourvu d'un pouvoir d'expression intéressant puisque comparable à celui de Prolog, dans lequel nous exprimons toutes les contraintes, et d'autre part, menant à des exécutions performantes, comparables à celles de solveurs SAT.

7.2 Extension du réseau de *E. coli*

En [14], les auteurs présentent un modèle instancié du réseau de gènes de *E.coli*, mais ce modèle ne reproduit pas l'ensemble des observations biologiques connues. En [13], ils étendent ce réseau pour respecter ces observations. Le nouveau réseau comporte trois gènes supplémentaires : $rpos$, $rssB$ et $gyrI$. Il comporte 9 gènes et 26 interactions. Les seuils de ces nouvelles interactions ne sont pas toujours connus, et notamment, ceux du gène $rpos$, qui agit sur trois autres gènes.

Nous avons modélisé ce réseau et adapté les contraintes imposées au réseau initial. Nous nous intéressons à l'existence d'une instanciation respectant l'ensemble des contraintes.

Si nous imposons que les chemins soient de longueur maximale 7, qui est la plus petite longueur possible, nous n'obtenons pas de modèles. Par contre, si nous permettons que les chemins soient de longueur 8, nous obtenons des modèles.

Dans le cas des chemins de longueur 7, la recherche du plus petit ensemble de contraintes à relâcher pour obtenir des modèles indique qu'il faut relâcher une contrainte parmi $\{C_{gyrAB1}, C_{topA1}, C_{gyrAB2}\}$, C_{gyrAB2} étant une contrainte exprimant que $K_{gyrAB}^{fisgyrI} \leq K_{gyrAB}^{fis}$, dont il reste à examiner la pertinence biologique. Ce résultat est obtenu en moins de cinq secondes (avec un Core 2 Duo 3GHz).

Dans ce réseau, le gène $rpos$ agit sur les gènes $gyrI$, $topA$ et $rssB$, mais, ne possédant aucune information sur les seuils de ces interactions, nous considérons que $rpos$ possède trois seuils, un pour chaque interaction. Nous nous intéressons au nombre minimum nécessaire de seuils de ce gène permettant la cohérence. Pour des chemins de longueur 8, nous trouvons que l'existence de seulement deux seuils suffit pour obtenir des modèles. Pour les chemins de longueur 7, nous obtenons les même résultats en relâchant une contrainte parmi $\{C_{gyrAB1}, C_{topA1}, C_{gyrAB2}\}$.

8 Conclusion

Nous avons présenté une nouvelle modélisation des réseaux de régulation génique utilisant le paradigme logique ASP et l'intérêt de ce paradigme pour ce faire. Le bien fondé de l'usage méthodique des règles ASP pour introduire les domaines de valeur des atomes nécessaires et ensuite celui des règles d'intégrité pour limiter l'ensemble des modèles possibles est illustré en 4. La représentation des hypothèses biologiques exposées en 5 montre l'importance de disposer de défauts normaux. Enfin, nous signalons en 6 comment, aussi à l'aide de défauts, éviter un processus para-logique dans le cas d'incohérence et comment inférer d'une façon très simple de nouvelles propriétés biologiques considérées comme des formules vraies dans tous les answer sets. Les aspects pratiques abordés en 7 confirment de bonnes performances sur des réseaux réels. Celles-ci dépendent évidemment du nombre n d'espèces et du demi-degré intérieur en moyenne f du graphe d'interaction. Les contraintes de succession sont proportionnelles à n et celles relatives à l'existence d'un chemin à la longueur du chemin. Le nombre de paramètres cinétiques est de l'ordre de 2^f, mais en général f est faible (vaut environ 3). Il reste que pour des réseaux de taille importante (plus de 20 espèces), une approche par décomposition/composition semble la plus appropriée, à la fois sur le plan des performances et de la compréhension de leur fonctionnement [3].

À notre connaissance, très peu d'équipes abordent l'analyse des réseaux de Thomas avec une approche de notre type [2][6]. Souvent, leurs travaux sont basés sur des outils de Model-Checking. Ces outils ont pour objectif de vérifier qu'un système de transition instancié satisfait des formules CTL (ou logique temporelle similaire). Aussi, de par cette origine, ils sont limités au regard des fonctionnalités présentées ici. Par exemple, du fait de la définition de CTL, il n'est pas possible d'exprimer l'existence de deux états stationnaires différents a priori non connus. Où encore, du fait de leur finalité, le relâchement automatique de contraintes ou la déduction de propriétés ne sont pas proposés par ces outils.

La technologie ASP est utilisée par ailleurs pour résoudre des problèmes d'inconsistance touchant aussi des réseaux biologiques [11][15]. Ces travaux se dis-

tinguent de ceux présentés ici dans la mesure où ils prennent seulement en compte l'aspect « statique » des réseaux : ils se consacrent à l'étude des interactions entre espèces biologiques et non pas à l'évolution temporelle d'un système biologique.

Nous nous intéressons maintenant à une formalisation étendue des réseaux biologiques intégrant des réactions métaboliques. Il s'agit de modéliser des réactions de nature différente dans un seul réseau. En effet, les réactions métaboliques possèdent des propriétés stœchiométriques que ne possèdent pas les interactions géniques. Ce formalisme étendu permet la modélisation de problèmes biologiques complexes comme l'homéostasie du fer dans les cellules mammifères. Un autre axe de recherche est l'aide au choix d'expériences.

Remerciements

Nous remercions M. Gebser et T. Schaub pour des échanges fructueux que nous avons eus et leurs conseils.

Ce travail a été soutenu par Microsoft Research à travers son programme de financement du doctorat de N.M.

Références

[1] Chitta Baral. *Knowledge Representation, Reasoning, and Declarative Problem Solving*. Cambridge University Press, New York, NY, USA, 2003.

[2] Gilles Bernot, Jean-Paul Comet, Adrien Richard, and Janine Guespin. Application of formal methods to biological regulatory networks : extending Thomas' asynchronous logical approach with temporal logic. *Journal of Theoretical Biology*, 229(3) :339 – 347, 2004.

[3] Gilles Bernot and Fariza Tahi. Behaviour preservation of a biological regulatory network when embedded into a larger network. *Fundam. Inf.*, 91 :463–485, August 2009.

[4] Philippe Besnard. *An Introduction to Default Logic*. Springer, 1989.

[5] Geneviève Bossu and Pierre Siegel. Saturation, nonmonotonic reasoning and the closed-world assumption. *Artif. Intell.*, 25 :13–63, January 1985.

[6] Nathalie Chabrier and François Fages. Symbolic model checking of biochemical networks. In *Computational Methods in Systems Biology*, Lecture Notes in Computer Science.

[7] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. *LNCS*, 131 :52–71, 1982.

[8] Fabien Corblin, Lucas Bordeaux, Eric Fanchon, Youssef Hamadi, and Laurent Trilling. Connections and integration with SAT solvers : A survey and a case study in computational biology. In Panos M. Pardalos, Pascal van Hentenryck, and Michela Milano, editors, *Hybrid Optimization*, volume 45 of *Optimization and Its Applications*, pages 425–461. Springer New York, 2011.

[9] Fabien Corblin, Éric Fanchon, and Laurent Trilling. Applications of a formal approach to decipher discrete genetic networks. *BMC Bioinformatics*, 11(1), 2010.

[10] Fabien Corblin, Sébastien Tripodi, Éric Fanchon, Delphine Ropers, and Laurent Trilling. A declarative constraint-based method for analyzing discrete genetic regulatory networks. *Biosystems*, 98 :91–104, 2009.

[11] Martin Gebser, Carito Guziolowski, Ivanchev M., Torsten Schaub, Anne Siegel, Philippe Veber, and Sven Thiele. Repair and Prediction (under Inconsistency) in Large Biological Networks with Answer Set Programming. In *Principles of Knowledge Representation and Reasoning : Proceedings of the Twelfth International Conference, KR 2010*, Toronto Canada, 2010. AAAI Press.

[12] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Max Ostrowski, Torsten Schaub, and Sven Thiele. A user's guide to gringo, clasp, clingo, and iclingo (version 3.x). Octobre 2010.

[13] Pedro T. Monteiro, Delphine Ropers, Radu Mateescu, Ana T. Freitas, and Hidde de Jong. Temporal logic patterns for querying dynamic models of cellulat interaction networks. *Bioinformatics*, 24 :227–233, 2008.

[14] Delphine Ropers, Hidde de Jong, Michel Page, Dominique Schneider, and Johannes Geiselmann. Qualitative simulation of the carbon starvation response in *escherichia coli*. *Biosystems*, 84(2) :124–152, 2006.

[15] Torsten Schaub and Sven Thiele. Metabolic network expansion with answer set programming. In Patricia Hill and David Warren, editors, *Logic Programming*, volume 5649 of *Lecture Notes in Computer Science*, pages 312–326. Springer Berlin / Heidelberg, 2009.

[16] René Thomas and Marcelle Kaufman. Multistationarity, the basis of cell differentiation and memory. II. logical analysis of regulatory networks in terms of feedback circuits. *CHAOS*, 11(1) :180–195, 2001.

Modélisation et résolution de problèmes de plus grands sous-graphes communs à l'aide de la programmation par contraintes

Samba Ndojh Ndiaye et Christine Solnon

Université de Lyon, CNRS
Université Lyon 1, LIRIS, UMR5205, F-69622, France
samba-ndojh.ndiaye@liris.cnrs.fr;christine.solnon@liris.cnrs.fr

Résumé

La distance entre deux graphes est généralement définie par rapport à la taille d'un plus grand sous-graphe commun. Ce sous-graphe commun peut être un sous-graphe induit, obtenu en supprimant des sommets, ou un sous-graphe partiel, obtenu en supprimant des arcs et des sommets. Dans cet article, nous modélisons ces deux problèmes dans un cadre unifié à l'aide de la contrainte souple *allDiff*. Nous introduisons également différents niveaux de propagation, et nous les comparons expérimentalement sur différents types de graphes.

Abstract

The distance between two graphs is usually defined by means of the size of a largest common subgraph. This common subgraph may be an induced subgraph, obtained by removing nodes, or a partial subgraph, obtained by removing arcs and nodes. In this paper, we introduce two soft CSPs which model these two maximum common subgraph problems in a unified framework. We also introduce and compare different CP models, corresponding to different levels of constraint propagation.

1 Introduction

Les graphes sont utilisés dans de nombreuses applications pour représenter des objets structurés tels que, par exemple, des molécules, des images ou des réseaux d'interactions biochimiques. Dans beaucoup de ces applications, il est nécessaire de disposer d'une mesure de distance entre deux graphes, distance qui est souvent définie par rapport à la taille du plus grand sous-graphe commun aux deux graphes. Plus précisément, on peut soit rechercher le plus grand sous-graphe induit (ayant le plus grand nombre de sommets), soit rechercher le plus grand sous-graphe partiel (ayant le plus grand nombre d'arcs). Ces deux problèmes sont NP-difficiles dans le cas général [7], et ont fait l'objet de nombreux travaux, notamment en bioinformatique et chemoinformatique [15, 13].

L'utilisation de la programmation par contraintes pour la résolution du problème d'isomorphisme de sous-graphes a été récemment étudiée dans [18, 16]. Ces travaux ont notamment montré l'intérêt de la contrainte globale *allDiff* pour résoudre ce genre de problème. De fait, cette contrainte a été initialement introduite par Régin pour résoudre ce problème [15].

Dans cet article, nous étudions comment utiliser la programmation par contraintes pour résoudre des problèmes de recherche de plus grands sous-graphes communs. Nous nous intéressons tout particulièrement à l'apport de la contrainte globale *allDiff* pour la résolution de ces problèmes. Dans la section 2, nous rappelons quelques définitions et nous décrivons les principales approches existantes. Dans la section 3, nous montrons comment modéliser ces deux problèmes en termes de contraintes dures et souples. Cette modélisation est faite dans un cadre unifié pour les deux problèmes. Dans la section 4, nous introduisons différents niveaux de propagation de contraintes. Dans la section 5, nous comparons expérimentalement ces différents modèles.

2 Contexte

2.1 Définitions

Un graphe $G = (N, A)$ se compose d'un ensemble de nœuds N et un ensemble d'arcs $A \subseteq N \times N$. Nous considérons implicitement des graphes orientés, dont

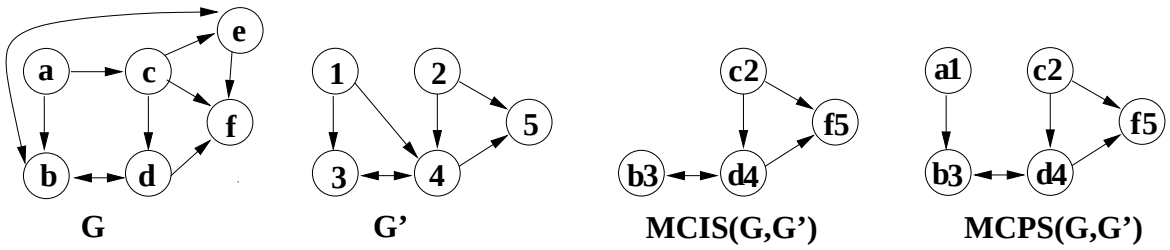


FIGURE 1 – Exemple de MCIS et MCPS pour deux graphes G et G'.

les arcs sont des couples orientés de nœuds. Les résultats présentés dans la suite peuvent être aisément généralisés aux graphes non orientés en associant à chaque arête $\{u, v\}$ deux arcs (u, v) et (v, u).

Soient deux graphes $G = (N, A)$ et $G' = (N', A')$. G est *isomorphe* à G' s'il existe une fonction bijective $f : N \to N'$ qui préserve les arcs, *i.e.*,

$$\forall (u, v) \in N \times N, (u, v) \in A \Leftrightarrow (f(u), f(v)) \in A'$$

Un *sous-graphe induit* est obtenu en supprimant des nœuds, *i.e.*, G' est un sous-graphe induit de G si $N' \subseteq N$ et $A' = A \cap N' \times N'$. Un *sous-graphe partiel* est obtenu en supprimant des nœuds et des arcs, *i.e.*, G' est un sous-graphe partiel de G si $N' \subseteq N$ et $A' \subseteq A \cap N' \times N'$.

Nous notons $G_{\downarrow S}$ le sous-graphe obtenu en gardant un sous-ensemble S de composants de G : si S est un sous-ensemble de nœuds (*i.e.*, $S \subseteq N$), alors $G_{\downarrow S}$ est le sous-graphe induit obtenu en gardant ces nœuds (*i.e.*, $G_{\downarrow S} = (S, A \cap S \times S)$) ; si S est un sous-ensemble d'arcs (*i.e.*, $S \subseteq A$), alors $G_{\downarrow S}$ est le sous-graphe partiel obtenu en gardant ces arcs (*i.e.*, $G_{\downarrow S} = (N_S, S)$ avec $N_S = \{u \in N \mid \exists v \in N, (u, v) \in S \vee (v, u) \in S\}$).

Un *sous-graphe commun* est un graphe isomorphe à des sous-graphes de G et G'. La similarité de deux graphes est généralement définie par rapport à la taille d'un sous-graphe commun [3] : plus ce sous-graphe est grand, plus les graphes sont similaires. La taille d'un sous-graphe est définie différemment selon que l'on considère des sous-graphes induits ou partiels : un *plus grand sous-graphe partiel (MCPS)* est un sous-graphe partiel qui a le plus grand nombre d'arcs, tandis qu'un *plus grand sous-graphe induit (MCIS)* est un sous-graphe induit qui a le plus grand nombre de sommets. La figure 1 montre un exemple de MCPS et MCIS.

2.2 Approches complètes existantes pour rechercher des MCIS et MCPS

La plupart des approches complètes pour rechercher des plus grands sous-graphes sont basées sur une reformulation du problème en un problème de recherche de clique maximum dans un graphe de compatibilité (dont les nœuds correspondent à des couples de nœuds pouvant être appariés, et les arêtes à des paires de couples de nœuds qui sont compatibles) [1, 5, 12].

McGregor [10] propose une approche différente basée sur un principe de Séparation et Evaluation (*Branch and Bound*) : chaque nœud de l'arbre de recherche correspond à l'appariement de deux composants, et la fonction d'évaluation estime le nombre de composants pouvant encore être appariés de sorte que la branche courante est coupée dès lors que cette borne devient inférieure à la taille du plus grand sous-graphe commun trouvé jusqu'ici.

Conte *et al* [4] ont comparé ces deux approches dans un même cadre expérimental sur une grande base de graphes. Ils ont montré qu'aucune approche ne surpasse l'autre : la meilleure approche change selon les caractéristiques des graphes.

Vismara et Valery [17] montrent comment rechercher des MCIS et MCPS à l'aide de la programmation par contraintes. Ils considèrent des cas particuliers de ces deux problèmes, où le sous-graphe recherché doit être connexe, et ils introduisent pour cela une contrainte globale de connexité. Ils utilisent des contraintes binaires de différence pour assurer que l'appariement calculé soit injectif. Ils comparent l'approche basée sur la programmation par contraintes avec une approche basée sur la recherche de cliques, et ils montrent que la programmation par contraintes obtient de meilleurs résultats.

3 Modélisation des problèmes de recherche de MCIS et MCPS à l'aide de CSP souples

Dans cette section, nous introduisons deux problèmes de satisfaction de contraintes (CSP) souples qui modélisent respectivement les problèmes de recherche du MCIS et du MCPS pour deux graphes $G = (N, A)$ et $G' = (N', A')$. Ces deux modèles diffèrent principalement sur leurs variables : pour le MCIS, les variables sont associées aux nœuds de G, alors que pour

le MCPS, les variables sont associées aux arcs. Dans les deux cas, la valeur affectée à la variable associée à un composant (nœud ou arc) de G correspond au composant avec lequel il est apparié dans G'. Comme certains composants peuvent ne pas être appariés, nous introduisons une valeur joker $\perp$ qui désigne le fait qu'un composant n'est pas apparié. Ainsi, pour le MCIS, le domaine de chaque variable x_u associée à un nœud $u \in N$ est $D(x_u) = N' \cup \{\perp\}$ tandis que pour le MCPS, le domaine de chaque variable x_{uv} associée à un arc $(u, v) \in A$ est $D(x_{uv}) = A' \cup \{\perp\}$.

Dans les deux cas, il existe deux types de contraintes. Un premier ensemble de contraintes binaires assure que les relations de voisinage définies par les arcs sont conservées. Pour le MCIS, ces contraintes binaires assurent que les relations d'adjacence entre les nœuds appariés sont conservées : étant données deux variables x_u et x_v, respectivement associées à des noeuds u et v de G, nous définissons

$$C_{arc}(x_u, x_v) \equiv \quad (x_u = \perp) \vee (x_v = \perp) \vee \\ ((u, v) \in A \Leftrightarrow (x_u, x_v) \in A')$$

Pour le MCPS, ces contraintes binaires assurent que les relations d'incidence entre les arcs appariés sont conservées : étant données deux variables x_{uv} et x_{wy} respectivement associées aux arcs (u, v) et (w, y) de G, nous définissons

$$C_{arc}(x_{uv}, x_{wy}) \equiv \quad (x_{uv} = \perp) \vee (x_{wy} = \perp) \vee \\ R((u, v), (w, y), x_{uv}, x_{wy})$$

où R est un prédicat qui vérifie que (u, v) et (w, y) ont les mêmes relations d'incidence que les arcs de G' affectés à x_{uv} et x_{wy}, *i.e.*, $R((u, v), (w, y), (u', v'), (w', y')) \equiv (u = v \Leftrightarrow u' = v') \wedge (u = w \Leftrightarrow u' = w') \wedge (u = y \Leftrightarrow u' = y') \wedge (v = w \Leftrightarrow v' = w') \wedge (v = y \Leftrightarrow v' = y') \wedge (w = y \Leftrightarrow w' = y')$.

Enfin, nous devons exprimer le fait que l'appariement doit être injectif (deux composants différents de G doivent être appariés à deux composants différents de G'). Ce genre de contrainte pourrait être modélisée avec une contrainte globale *allDiffExcept⊥(X)* qui oblige toutes les variables de X à prendre des valeurs distinctes, sauf si elles sont affectées à la valeur joker $\perp$ [2]. Pour trouver un plus grand sous-graphe commun, nous cherchons un appariement partiel injectif qui maximise le nombre de composants appariés, *i.e.*, nous devons minimiser le nombre de variables affectées à $\perp$. Cela pourrait être réalisé par l'ajout d'une contrainte *atmost*$(b - 1, X, \perp)$ à chaque fois qu'une solution cohérente σ est trouvée, où b est le nombre de variables affectées à $\perp$ dans σ. Cependant, ce modèle sépare l'évaluation de la fonction de coût de la contrainte globale *allDiff*.

Les contraintes globales d'optimisation permettent généralement un meilleur filtrage en liant les contraintes aux variables de coût à optimiser, comme cela a été proposé dans [6]. En particulier, la contrainte souple *allDiff* [11] lie un ensemble X de variables à une variable additionnelle *cost*, qui est contrainte à être égale au nombre de variables de X dont on devrait changer la valeur pour satisfaire la contrainte globale *allDiff(X)*, et qui doit être minimisée (nous considérons que les coûts de violation sont basés sur les variables).

Pour trouver un appariement partiel injectif qui minimise le nombre de composants non appariés, nous introduisons une variable supplémentaire, $x_\perp$, dont le domaine est $D(x_\perp) = \{\perp\}$, et nous ajoutons une contrainte souple *allDiff*$(X \cup \{x_\perp\}, cost)$. Notons que $x_\perp$ est toujours affectée à $\perp$: cette variable garantit que toutes les autres variables sont affectées à des valeurs différentes de $\perp$ à chaque fois que cela est possible (*i.e.*, quand G et G' sont isomorphes).

Nous pouvons maintenant définir les deux CSP souples modélisants les problèmes de recherche de MCIS et de MCPS.

MCIS. Nous définissons le CSP :
- Variables : $X = \{x_u \mid u \in N\} \cup \{x_\perp, cost\}$
- Domaines : $D(x_\perp) = \{\perp\}$, $D(cost) = \{0, \ldots, |N|\}$ et, $\forall u \in N, D(x_u) = N' \cup \{\perp\}$
- Contraintes dures : $\forall \{u, v\} \subseteq N, C_{arc}(x_u, x_v)$
- Contrainte souple : *allDiff* $(X, cost)$

MCPS. Nous définissons le CSP :
- Variables : $X = \{x_{uv} \mid (u, v) \in A\} \cup \{x_\perp, cost\}$
- Domaines : $D(x_\perp) = \{\perp\}$, $D(cost) = \{0, \ldots, |A|\}$ et, $\forall (u, v) \in A, D(x_{uv}) = A' \cup \{\perp\}$
- Contraintes dures : $\forall \{(u, v), (w, y)\} \subseteq A, C_{arc}(x_{uv}, x_{wy})$
- Contrainte souple : *allDiff* $(X, cost)$

Calcul d'un plus grand sous-graphe commun à partir d'une solution d'un CSP souple. Une solution est une affectation σ des variables de X qui satisfait toutes les contraintes dures et qui minimise le coût de violation de la contrainte souple de sorte que $\sigma(cost)$ soit égal à ce coût de violation. Soit $\sigma(X) \setminus \perp$ l'ensemble des valeurs différentes de $\perp$ qui sont affectées à des variables de X. Nous pouvons facilement vérifier que, pour le MCIS (resp. MCPS), $G'_{\downarrow \sigma(X) \setminus \perp}$ est isomorphe à un sous-graphe induit (resp. partiel) de G.

Notons que nous ne pouvons pas définir le sous-graphe commun induit en gardant simplement chaque nœud de G dont la variable associée a été affectée à une valeur différente de $\perp$. En effet, quand plusieurs nœuds de G ont le même voisinage, il peut arriver que

les variables associées à ces nœuds soient affectées à la même valeur (différente de $\perp$).

Considérons par exemple les graphes de la figure 1. Pour le MCIS, l'affectation $\sigma = \{x_\perp = \perp, x_a = \perp, x_b = 3, x_d = 4, x_c = 2, x_f = 5, x_e = 4\}$ est une solution optimale. Dans ce cas, $\sigma(X) \setminus \perp = \{2, 3, 4, 5\}$ et $G'_{\downarrow\sigma(X)\setminus\perp}$ est le sous-graphe obtenu en supprimant le nœud 1 de G'. Dans cette solution, à la fois x_d et x_e sont affectées à 4 car d et e ont le même voisinage.

La taille du plus grand sous-graphe commun $G'_{\downarrow\sigma(X)\setminus\perp}$ est égale à $c - \sigma(cost)$ où c est le nombre de composants de G ($c = |N|$ pour le MCIS et $c = |A|$ pour le MCPS), et $\sigma(cost)$ est la valeur de la variable *cost* dans la contrainte souple *allDiff*. Comme la valeur de *cost* est minimale, la taille de $G'_{\downarrow\sigma(X)\setminus\perp}$ est maximale.

Sur notre exemple précédent, nous avons $\sigma(cost) = 2$ et $|N| = 6$ de sorte que $G'_{\downarrow\sigma(X)\setminus\perp}$ a 4 nœuds.

Extension aux graphes étiquetés. Dans les graphes étiquetés, les nœuds et les arcs sont associés à des étiquettes. Dans ce cas, le sous-graphe commun doit apparier des composants ayant des étiquettes identiques. Ce type de contrainte est facile à intégrer dans notre modèle. Pour le MCIS, nous limitons le domaine de chaque variable x_u aux nœuds qui ont la même étiquette que u, et nous assurons que les étiquettes des arcs sont préservées dans les contraintes C_{arc}. Pour le MCPS, nous limitons le domaine de chaque variable x_{uv} aux arcs qui ont la même étiquette que (u, v) et dont les extrémités ont les mêmes étiquettes.

4 Propagation de contraintes

Les deux CSP introduits dans la section précédente sont très similaires : ils combinent tous deux un ensemble de contraintes binaires dures avec une contrainte souple *allDiff*. Nous proposons de considérer différents niveaux de propagation de ces contraintes.

4.1 Propagation de la contrainte souple allDiff

Nous considérons 3 niveaux de propagation.

GAC(allDiff). La plus forte propagation, notée *GAC(allDiff)*, assure la cohérence d'arc généralisée comme cela est proposé dans [11]. Plus précisément, nous recherchons un couplage maximum dans le graphe biparti $G_b = (X, V, E_b)$ où X est l'ensemble des variables, V est l'ensemble des valeurs dans les domaines des variables, et E_b est l'ensemble des arêtes $(x, v) \in X \times V$ tel que $v \in D(x)$. Un couplage de G_b est un sous-ensemble d'arêtes de E_b dont tous les sommets sont différents. La cardinalité d'un couplage maximum de G_b donne le nombre maximum de variables qui peuvent être affectées à des valeurs différentes et, par conséquent, fournit une borne supérieure pour *cost*. Lorsque la taille du couplage maximum de G_b est supérieure à la borne inférieure de *cost*, nous ne pouvons pas filtrer les domaines des variables. Cependant, dès que le couplage maximum de G_b est aussi grand que la borne inférieure de *cost*, nous pouvons filtrer les domaines en cherchant les arêtes de G_b n'appartenant à aucun couplage maximum (voir [14, 11] pour plus de détails).

Comme proposé dans [11, 14, 8], nous utilisons l'algorithme de [9] pour calculer un couplage maximum, et nous exploitons le fait que cet algorithme est incrémental : à chaque nœud de l'arbre de recherche, nous mettons à jour le dernier couplage calculé en enlevant les arêtes correspondant aux valeurs filtrées, et nous complétons ce couplage jusqu'à ce qu'il devienne maximum.

bound(cost)+FC(diff). Nous avons également considéré un filtrage moins fort, noté bound(cost)+FC(diff). Ce filtrage n'assure pas la cohérence d'arc généralisée, mais vérifie simplement qu'il existe un couplage du graphe biparti G_b dont la taille est supérieure ou égale à la borne inférieure de *cost*. Cette vérification est faite de façon incrémentale et fainéante : à chaque nœud de l'arbre de recherche, après avoir mis à jour le dernier couplage calculé, nous ne cherchons à l'étendre que si sa taille est strictement inférieure à la borne inférieure de *cost*. Cette vérification de borne est combinée à une simple vérification en avant (*forward-checking*) de la décomposition binaire de la contrainte globale *allDiff* : à chaque fois qu'une variable est affectée à une valeur v différente de $\perp$, nous enlevons v des domaines des variables non affectées.

FC(diff). Enfin, la propagation la plus faible, notée FC(diff), est une simple vérification en avant de la décomposition binaire de la contrainte *allDiff* qui enlève la valeur v des domaines des variables à chaque fois que v est affectée à une variable. La borne supérieure de la variable *cost* est mise-à-jour à chaque fois qu'une variable est affectée à $\perp$.

4.2 Propagation des contraintes dures C_{arc}

Tant que le domaine de la variable *cost* n'a pas été réduit à un singleton par la propagation de la contrainte souple *allDiff*, la valeur joker $\perp$ appartient au domaine de chaque variable non affectée. Dans ce cas, une vérification en avant des contraintes dures C_{arc} assure la cohérence d'arc de ces contraintes. En

effet, pour chaque paire de variables non affectées (x_i, x_j), et pour chaque valeur $v \in D(x_i)$, la valeur $\perp$ appartient à $D(x_j)$ et est un support de v car $C_{arc}(x_i, x_j)$ est satisfaite dès lors que x_i ou x_j sont affectées à $\perp$.

Dès lors que le domaine de la variable *cost* a été réduit à un singleton, $\perp$ est supprimé des domaines des variables non affectées. Dans ce cas, le maintien de la cohérence d'arcs (MAC) peut permettre de filtrer plus de valeurs qu'une simple vérification en avant. Par conséquent, nous avons considéré deux niveaux de propagation :

- FC(C_{arc}) effectue une simple vérification en avant des contraintes C_{arc} ;
- MAC(C_{arc}) maintient la cohérence d'arc de ces contraintes (cependant, comme la vérification en avant assure la cohérence d'arc tant que $\perp$ n'a pas été supprimé des domaines, nous n'effectuons qu'une vérification en avant quand $\perp$ appartient aux domaines, et nous ne maintenons effectivement la cohérence d'arc qu'à partir du moment ou $\perp$ a été enlevé des domaines).

5 Résultats expérimentaux

Modèles comparés. Nous comparons les cinq modèles suivants :

- FC = FC(C_{arc}) + FC(diff) ;
- FC+bound = FC(C_{arc})+bound(cost)+FC(diff) ;
- FC+GAC = FC(C_{arc}) + GAC(allDiff) ;
- MAC+bound = MAC(C_{arc}) + bound(cost) + FC(diff) ;
- MAC+GAC = MAC(C_{arc}) + GAC(allDiff).

En terme de puissance de filtrage, nous avons les relations suivantes : FC < FC+bound < FC+GAC < MAC+GAC et FC+bound < MAC+bound < MAC+GAC. En revanche, FC+GAC et MAC+bound ne sont pas comparables.

Notons que le modèle FC correspond en fait à l'approche par séparation et évaluation proposée par McGregor dans [10], ainsi qu'au modèle proposé dans [17] (excepté le fait que, dans [17], une contrainte de connectivité est ajoutée afin de garantir que les sous-graphes trouvés sont bien connexes).

Pour les 5 modèles, nous utilisons l'heuristique d'ordre *minDom* pour le choix des variables, et les valeurs sont choisies par ordre croissant. Les 5 modèles ont été programmés en C.

Jeux d'essais. Nous considérons une base de graphes générés aléatoirement et décrits dans [4]. Pour chaque graphe de la base, il existe trois étiquetages différents tels que le nombre d'étiquettes différentes est égal à 33%, 50% ou 75% du nombre de nœuds.

Nous comparons les résultats expérimentaux pour trois jeux d'essais de difficultés croissantes :

- Le jeu d'essai 1 recherche des MCIS dans des graphes orientés et étiquetés tels que le nombre d'étiquettes différentes est égal à 33% du nombre de nœuds (quand on augmente ce ratio, le problème devient beaucoup plus facile).
- Le jeu d'essai 2 recherche également des MCIS, mais dans des graphes non orientés et non étiquetés. Dans ce cas, les domaines des variables sont plus grands (car tous les sommets sont compatibles du fait de l'absence d'étiquettes), et les contraintes C_{arc} sont moins fortes.
- Le jeu d'essai 3 recherche des MCPS dans des graphes orientés et non étiquetés. Dans ce cas, les domaines des variables contiennent tous les arcs de G' et sont donc de plus grande taille que dans les deux cas précédents.

Nous avons adapté la taille des graphes en fonction de la difficulté de ces jeux d'essai de sorte que les problèmes puissent être résolus dans une limite de temps acceptable : pour le jeu d'essai 1 (resp. 2 et 3), nous considérons des graphes ayant 40 (resp. 30 et 20) nœuds.

Pour chaque jeu d'essai, nous donnons les résultats obtenus pour différentes classes de graphes :

- des graphes connexes générés aléatoirement dont la connectivité est $\eta \in \{0.05, 0.2\}$ (r005 et r02) ;
- des maillages réguliers 2D, 3D et 4D (m2D, m3D et m4D) :
- des maillages irréguliers 2D, 3D et 4D, avec $\rho = 0.6$ (m2Dr, m3Dr, et m4Dr) ;
- des graphes réguliers dont le degré est borné, avec $V \in \{3, 9\}$ (b03 et b09) ;
- des graphes irréguliers dont le degré est borné, avec $V \in \{3, 9\}$ (b03m et b09m).

Chaque classe contient 150 paires de graphes correspondant aux 30 premières instances pour chacune des 5 tailles possibles de plus grand sous-graphes (*i.e.*, 10%, 30%, 50%, 70% et 90% du nombre de nœuds).

Le tableau 1 compare les 5 modèles sur les 3 jeux d'essais. Nous constatons que le modèle FC (correspondant aux approches proposées dans [10] et [17]) est clairement dépassé par tous les autres modèles qui effectuent des filtrages plus forts. En effet, FC effectue un bornage passif de la variable *cost*, en comptant simplement le nombre de variables devant être affectées à $\perp$. Tous les autres modèles ont un mécanisme de calcul de borne actif qui vérifie que le nombre de variables pouvant être affectées à des valeurs différentes est suffisamment grand.

Le calcul de bornes fainéant (bound(cost)) introduit dans la section 4 réduit de façon drastique l'espace de recherche et les temps de FC+bound sont toujours si-

		FC			FC+bound			FC+GAC			MAC+bound			MAC+GAC		
		%S	temps	#Kn	%S	temps	#Kn	%S	temps	#Kn	%S	temps	#Kn	%S	temps	#Kn
Jeu d'essai 1	b03	100	105.79	56074	100	20.81	6066	100	24.83	4831	100	**13.43**	3166	100	13.86	1502
	b03m	100	143.74	80297	100	26.17	7754	100	28.86	5651	100	16.91	4021	100	**15.06**	1742
	b09	100	0.11	50	100	0.07	19	100	0.08	17	100	**0.06**	15	100	0.08	10
	b09m	100	0.12	54	100	0.08	22	100	0.09	20	100	**0.07**	17	100	0.10	11
	m2D	100	98.62	53960	100	18.05	5200	100	20.19	3664	100	12.25	2876	100	**10.94**	1220
	m2Dr	100	8.06	3990	100	3.14	864	100	3.65	724	100	**2.21**	523	100	2.38	291
	m3D	100	15.05	7532	100	5.55	1536	100	5.82	1157	100	**3.69**	865	100	4.86	439
	m3Dr	100	3.90	1913	100	1.62	419	100	1.82	359	100	1.14	273	100	**1.13**	154
	m4D	100	97.33	50940	100	12.09	3147	100	12.94	2496	100	**8.17**	1832	100	9.28	835
	m4Dr	100	5.85	2745	100	1.87	471	100	2.04	387	100	**1.38**	325	100	1.50	169
	r005	100	19.47	10540	100	4.72	1295	100	5.68	1040	100	**3.17**	741	100	3.57	393
	r02	100	0.02	10	100	0.02	6	100	0.02	5	100	**0.01**	4	100	0.02	3
Jeu d'essai 2	b03	72	756.25	312080	100	**68.87**	10256	100	77.93	7728	97	212.41	3679	98	231.77	2301
	b03m	57	1081.52	441952	100	**101.77**	14749	100	121.99	12043	97	343.77	6017	97	397.14	4010
	b09	100	147.80	62050	100	**35.09**	7709	100	40.27	6699	100	41.49	6068	100	44.69	3531
	b09m	99	342.89	149613	100	**86.07**	20054	100	94.98	16364	100	101.07	15347	100	103.71	8439
	m2D	61	985.35	394241	100	**103.17**	16003	100	131.48	13532	96	365.66	7582	95	411.89	4491
	m2Dr	62	998.30	383680	100	**171.70**	29757	100	201.49	24344	99	428.35	17228	98	482.21	9128
	m3D	76	737.81	277429	100	**81.78**	13206	100	101.71	11570	100	240.58	7538	98	284.13	4331
	m3Dr	83	681.18	266386	100	**115.49**	21872	100	156.56	20579	100	254.37	13715	100	316.93	8262
	m4D	46	1276.08	498405	100	**120.71**	17386	100	129.53	13257	100	360.14	8699	96	423.83	5704
	m4Dr	50	1448.08	549375	100	**165.51**	27849	100	195.86	23127	100	421.02	16238	100	467.62	8966
	r005	50	1236.75	515935	99	**142.04**	22122	98	175.62	17625	93	443.26	8601	92	494.76	5099
	r02	100	474.12	222831	100	246.16	67044	100	283.70	58036	100	**238.91**	53792	100	242.84	32445
Jeu d'essai 3	b03	100	34.44	9364	100	8.67	1178	100	10.93	1163	100	**7.13**	582	100	8.37	392
	b03m	100	47.41	13809	100	9.62	1350	100	10.41	1172	100	7.33	700	100	**7.04**	386
	b09	0	-	-	0	-	-	0	-	-	0	-	-	0	-	-
	b09m	0	-	-	0	-	-	0	-	-	0	-	-	0	-	-
	m2D	100	214.00	59029	100	32.17	3933	100	33.69	3324	100	26.16	2159	100	**25.83**	1165
	m2Dr	0	-	-	0	-	-	0	-	-	0	-	-	0	-	-
	m3D	90	1122.39	263519	100	206.22	23399	100	282.27	24543	100	**187.90**	14170	100	226.57	9000
	m3Dr	0	-	-	0	-	-	0	-	-	0	-	-	0	-	-
	r005	100	11.28	4333	100	2.12	354	100	2.56	358	100	**1.39**	144	100	1.56	98
	r02	0	-	-	0	-	-	0	-	-	0	-	-	0	-	-

TABLE 1 – Comparaison des 5 modèles sur les 3 jeux d'essais. Chaque ligne donne successivement le nom de la classe et, pour chaque modèle, le pourcentage d'instances (%S) résolues en moins de 30mn sur un 2.26 GHz Intel Xeon E5520, le temps CPU en secondes (temps) et le nombre de milliers de nœuds de l'arbre de recherche (#Kn). Le temps CPU ainsi que le nombre de nœuds sont des résultats moyens sur les instances résolues uniquement.

gnificativement plus bas que ceux de FC. GAC(allDiff) réduit encore plus l'espace de recherche mais la différence est moins forte de sorte que les temps CPU de FC+GAC sont supérieurs à ceux de FC+bound. Cette tendance est observée sur les trois jeux d'essais.

Le fait de remplacer FC(C_{arc}) par MAC(C_{arc}) réduit fortement le nombre de nœuds explorés, mais ce filtrage a aussi un coût plus important de sorte qu'il ne permet pas toujours de réduire le temps CPU : il améliore les performances sur les jeux d'essais 1 et 3, mais les détériore sur le jeu d'essai 2. Ainsi, les meilleures approches sont MAC+bound et MAC+GAC pour les jeux d'essais 1 et 3 tandis que la meilleure approche est FC+bound pour le jeu d'essai 2. En fait, les jeux d'essais 1 et 3 (tels que les graphes sont orientés) sont plus contraints que ceux du jeu d'essai 2, ce qui permet à MAC de filtrer plus de valeurs.

6 Conclusion

Nous avons montré comment modéliser des problèmes de recherche de plus grands sous-graphes communs sous la forme de problèmes de satisfaction de contraintes mêlant des contraintes dures, permettant d'assurer que les relations d'adjacence définies par les arcs sont conservées dans le sous-graphe commun, à une contrainte *allDiff* souple, permettant de maximiser la taille du sous-graphe commun. Nous avons considéré deux variantes du problème : la recherche d'un plus grand sous-graphe commun induit, où il s'agit de maximiser le nombre de sommets du sous-graphe, et la recherche d'un plus grand sous-graphe partiel, où il s'agit de maximiser le nombre d'arcs du sous-graphe.

Nous avons également introduit un nouvel algorithme pour la propagation de la contrainte souple *allDiff* : cet algorithme n'assure pas la cohérence d'arc généralisée mais vérifie simplement que le nombre de

variables pouvant être affectées à des valeurs différentes est supérieur ou égal à la borne inférieure de la variable de coût associée à la contrainte souple *allDiff*. Cette vérification est faite de façon incrémentale et fainéante, ce qui permet une propagation beaucoup plus rapide que la cohérence d'arc généralisée. Les résultats expérimentaux montrent que cette propagation fainéante assure un bon compromis entre force de filtrage et temps de propagation.

Les travaux futurs concerneront l'étude d'heuristiques d'ordre pour ces problèmes. En effet, les heuristiques d'ordre sont généralement très importantes pour améliorer l'efficacité de la résolution de problèmes d'optimisation, l'objectif étant de guider la recherche vers les meilleures solutions en premier de façon à pouvoir couper plus de branches. Nous souhaitons également effectuer plus d'expérimentations, sur des graphes de différentes natures, afin d'identifier les techniques de propagation les mieux adaptées en fonction de la nature des graphes considérés.

Références

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

[2] N. Beldiceanu, M. Carlsson, S. Demassey, and Thierry Petit. Global constraint catalog : Past, present and future. *Constraints*, 12(1) :21–62, 2007.

[3] H. Bunke and K. Sharer. A graph distance metric based on the maximal common subgraph. *Pattern Recognition Letters*, 19(3) :255–259, 1998.

[4] D. Conte, P. Foggia, and M. Vento. Challenging complexity of maximum common subgraph detection algorithms : A performance analysis of three algorithms on a wide database of graphs. *Graph Algorithms and Applications*, 11(1) :99–143, 2007.

[5] P. J. Durand, R. Pasari, J. W. Baker, and C. Tsai. An efficient algorithm for similarity analysis of molecules. *Internet Journal of Chemistry*, 2, 1999.

[6] F. Focacci, A. Lodi, and M. Milano. Cost-based domain filtering. In *CP'99*, 1999.

[7] M. R. Garey and D. S. Johnson. Computer and intractability. In *Freeman*, 1979.

[8] I. Gent, I. Miguel, and P. Nightingale. Generalised arc consistency for the alldiff constraint : An empirical survey. *Artificial Intelligence*, 172(18) :1973–2000, 2008.

[9] J. E. Hopcroft and R. M. Karp. An n5/2 algorithm for maximum matchings in bipartite graphs. *SIAM Journal on Computing*, 2(4) :225–231, 1973.

[10] J. J. McGregor. Backtrack search algorithms and the maximal common subgraph problem. *Software Practice and Experience*, 12(1) :23–34, 1982.

[11] T. Petit, J.-C. Régin, and C. Bessière. Specific filtering algorithms for over-constrained problems. In *CP'01, LNCS 2239*, pages 451–464. Springer, 2001.

[12] J. W. Raymond, E. J. Gardiner, and P. Willett. Calculation of graph similarity using maximum common edge subgraphs. *The Computer Journal*, 45(6), 2002.

[13] J. W. Raymond and P. Willett. Maximum common subgraph isomorphism algorithms for the matching of chemical structures. *Journal of computeraided molecular design*, 16(7) :521–533, 2002.

[14] J.-C. Régin. A filtering algorithm for constraints of difference in csps. In *AAAI-94*, pages 362–367, 1994.

[15] J.-C. Régin. *Développement d'Outils Algorithmiques pour l'Intelligence Artificielle. Application à la Chimie Organique*. PhD thesis, Université Montpellier II, 1995.

[16] C. Solnon. Alldifferent-based filtering for subgraph isomorphism. *Artificial Intelligence*, 174(12-13) :850–864, 2010.

[17] P. Vismara and B. Valery. Finding maximum common connected subgraphs using clique detection or constraint satisfaction algorithms. *Communications in Computer and Information Science*, 14(1) :358–368, 2008.

[18] S. Zampelli, Y. Deville, and C. Solnon. Solving subgraph isomorphism problems with constraint programming. *Constraints*, 15(3) :327–353, 2010.

Au-delà des produits cartésiens de domaines : l'exemple des octogones

Marie Pelleau, Charlotte Truchet, Frédéric Benhamou

Laboratoire d'Informatique de Nantes-Atlantique (LINA), UMR 6241
Université de Nantes, CNRS
2 rue de la Houssinière, Nantes, France
firstName.lastName@univ-nantes.fr

Résumé

Cet article propose une nouvelle représentation pour les domaines des variables continues s'inspirant des travaux de l'Interprétation Abstraite (IA). Il existe de nombreux domaines abstraits et à l'inverse des domaines en Programmation par Contraintes (PPC) certains prennent en compte des dépendances linéaires entre les variables. Les domaines abstraits peuvent être vus comme une implantation de sous-langages de contraintes. Nous montrons que les outils basiques de résolution en PPC (consistance, split, ...) se traduisent naturellement sur les domaines abstraits. Nous illustrerons cette adaptation avec un exemple détaillé, celui des octogones. Les octogones font partie de la famille des domaines abstraits faiblement relationnels (non cartésien) et sont une restriction des polyèdres où seules les contraintes de la forme $\pm V_i \pm V_j \leq c$ sont autorisées.

1 Introduction

Pour les problèmes à variables continues, le formalisme des contraintes par intervalles repose sur une encapsulation des réels dans l'ensemble des intervalles à borne flottante. Le manque de précision des intervalles peut générer une surapproximation des solutions (*e.g.* plusieurs occurrences d'une même variable, contraintes non linéaires). Plusieurs méthodes ont été proposées pour réduire cette surestimation utilisant des évaluateurs pour les contraintes non linéaires, une relaxation sûre des contraintes quadratiques, ou plus récemment la monotonie des contraintes.

Plutôt que d'essayer d'améliorer les méthodes par intervalles, nous avons décidé d'étudier l'utilisation des domaines abstraits de l'Interprétation Abstraite (IA) comme représentation en Programmation par Contraintes (PPC). L'IA traite de très gros programmes et repose notamment sur la notion de domaines abstraits, qui sont des surapproximations des traces de programmes, pouvant être relationnels (non cartésien). L'idée est d'utiliser ces domaines abstraits en PPC afin d'avoir une représentation plus précise que les intervalles lors de la résolution de problèmes continus. Les domaines des variables étant plus réduits, les solutions sont plus rapidement trouvées. En d'autres termes, améliorer la contraction pour avoir une résolution plus rapide. Afin de résoudre un problème avec une représentation des domaines différente, il est nécessaire de redéfinir certains outils basiques de la PPC. Ce papier donne des définitions générales pour ces outils.

Afin de répondre à ces questions, nous présenterons section 2, une manière générique d'étendre les domaines abstraits à la PPC. Section 3, nous illustrerons cette adaptation avec un exemple détaillé de domaine abstrait, celui des octogones. Nous définirons les outils nécessaires à la résolution de problèmes continus à l'aide des octogones, tel que la consistance octogonale. La section 4 présente les résultats obtenus avec les octogones.

1.1 Notations et rappels

On se contentera de rappeler les notations et définitions nécessaires pour la suite.

On considère un Problème de Satisfaction de Contraintes (CSP) sur les variables $\mathcal{V} = (V_1...V_n)$, prenant leurs valeurs dans les domaines $\mathcal{D} = (D_1...D_n)$, et avec les contraintes $(C_1...C_p)$. L'ensemble des instanciations possibles pour les variables est $D = D_1 \times ... \times D_n$. Les solutions du CSP sont les éléments de D qui satisfont les contraintes. On note $\mathcal{S}$ l'ensemble des solutions, $\mathcal{S} = \{(v_1...v_n) \in \mathcal{D}, \forall i \in \{1 \dots p\}, C_i(v_1...v_n)\}$,

et $\mathcal{S}_{C_i}$ les solutions pour la contrainte C_i.

Définition Soient $V_1...V_n$ des variables de domaines discrets finis $D_1...D_n$, et C une contrainte.
Les domaines $D_1...D_n$ sont dits *arc-consistants généralisés* (GAC) pour C ssi $\forall i \in \{1...n\}, \forall v \in D_i, \forall j \neq i, \exists v_j \in D_j$ tel que $C(v_1, v_2, ..., v_{i-1}, v, v_{i+1}, ...v_n)$.
Les domaines sont dits *bornes-consistants* (BC) pour C ssi $\forall i \in \{1...n\}$, D_i est un intervalle entier, $D_i = \{a_i...b_i\}$ avec $a_i, b_i \in \mathbb{N}$, et la condition précédente tient pour $v = a_i$ and $v = b_i$.

Définition Soient $V_1...V_n$ des variables de domaines continus représentés par les intervalles $D_1...D_n \in \mathbb{I}$, et C une contrainte. Les domaines $D_1...D_n$ sont dits *Hull-consistant* ssi $D_1 \times ... \times D_n$ est la plus petite boite flottante contenant les solutions.

La prochaine partie introduit brièvement l'IA et les notions utiles afin d'utiliser les domaines abstraits en PPC. Pour une présentation plus complète voir [6]

1.2 Interprétation Abstraite

L'IA est un domaine de la sémantique traitant la preuve de programmes. La correction de programmes ne pouvant être prouvée génériquement, l'IA met en place des méthodes permettant d'analyser automatiquement certaines propriétés d'un programme. Cette analyse est appelée *statique* car elle est effectuée au moment de la compilation. L'analyse repose sur l'approximation de toutes les exécutions possibles d'un programme. Si l'intersection de cette approximation et des zones dites dangereuses est vide, on sait que le programme est correct.

L'idée principale est d'abstraire les valeurs des variables. Plus précisément, l'ensemble des valeurs prises par une variable tout au long d'une exécution, appelé *sémantique concrète*, est remplacé par une approximation, appelée *sémantique abstraite*. Par exemple, les valeurs possibles pour une variable flottante peuvent être remplacées par un intervalle à bornes flottantes les contenant. Plusieurs opérateurs sont définis sur les domaines abstraits permettant de modéliser les effets des instructions du programme sur les variables. Cela entraine souvent une perte de précision notamment lors de l'analyse d'une boucle.

L'outil théorique sur lequel repose les domaines abstraits est la notion de treillis :

Définition Une relation $\sqsubseteq$ sur un ensemble non vide E est un *ordre partiel* (po ou poset) ssi elle est réflexive, antisymétrique et transitive. Un ordre partiel $(E, \sqsubseteq)$ est un *treillis* ssi pour $a, b \in E$, le couple $\{a, b\}$ a une plus petite borne supérieure et une plus grande borne inférieure. Il est *complet* ssi tout sous-ensemble a une plus petite borne supérieure.

Un treillis complet a un plus petit élément et un plus grand élément. Les domaines abstraits doivent être des treillis ou un ordre partiel complet. Une caractéristique importante des domaines abstraits est qu'ils sont reliés par des connexions de Gallois :

Définition Soient deux domaines abstraits E_1 et E_2 (chacun étant un po), une connexion de Gallois est définie par deux morphismes $\alpha : E_1 \to E_2$ et $\gamma : E_2 \to E_1$ tels que

- α et γ sont monotones,
- $\forall X_1 \in E_1, X_2 \in E_2, \alpha(X_1) \sqsubseteq X_2 \iff X_1 \sqsubseteq \gamma(X_2)$.

Remarque Les connexions de Gallois ne perdent pas d'éléments : $(\alpha \circ \gamma)(X_2) \sqsubseteq X_2$ et $X_1 \sqsubseteq (\gamma \circ \alpha)(X_1)$.

De nombreux domaines abstraits de formes différentes ont été définis. Ils peuvent prendre en compte plusieurs variables. Ils peuvent être non-relationnels, c'est-à-dire être un produit cartésien de domaines abstraits. Le produit cartésien d'intervalles est un exemple de domaine abstrait non-relationnel.

Mais ils peuvent aussi être relationnels, *e.g.* non cartésiens, comme le domaine abstrait Polyèdre qui prend en compte des relations linéaires entre les variables. Définir une connexion de Gallois entre un polyèdre et un domaine concret revient à résoudre des contraintes linéaires. Un autre exemple de domaine abstrait relationnel est celui des Octogones, une restriction des polyèdres, où seules les contraintes de la forme $\pm V_i \pm V_j \leq c$ avec c une constante, sont autorisées. Ce domaine est moins précis mais moins coûteux en terme de temps de calcul [9]. Beaucoup d'autre domaines abstrait ont été définis et étudiés, voir [8]. La raison pour laquelle il existe tant de domaines abstraits est la recherche d'un compromis entre précision et temps de calcul : un domaine abstrait précis offre une analyse plus fine mais requiert un plus grand temps de calcul.

La PPC et l'IA sont deux domaines différents, pourtant dans ces deux domaines les théories sous-jacentes reposent sur des notions de treillis et de point fixe. De plus, ces deux domaines calculent une sur-approximation d'un ensemble, à la différence qu'en IA on ne cherche pas toujours à être le *plus* précis possible mais à être *assez* précis afin d'éviter des zones dangereuses. En pratique le point fixe est rarement atteint en IA, ce qui permet d'analyser de très gros programmes. Notre idée est de profiter des points communs pour utiliser les domaines abstraits en PPC, et ainsi pouvoir s'abstraire des produits cartésiens lors de la résolution de problèmes continus.

2 Domaines abstraits pour la PPC

Un domaine abstrait peut être vu comme un sous-langage de contraintes permettant de définir des formes géométriques (*e.g.* les contraintes linéaires définissent les polyèdres). Des solveurs spécifiques sont utilisés pour implanter les domaines abstraits (*e.g.* la programmation linéaire pour les polyèdres). Ces sous-langages, adaptés à la PPC, peuvent calculer une approximation complète des solutions si une consistance dédiée est définie. En IA, le choix du domaine abstrait dépend des propriétés du programme à analyser [5]. La conclusion de [5], est qu'il faut considérer plusieurs domaines abstraits en même temps pour permettre un passage à l'échelle. Ainsi chaque domaine abstrait traite les expressions pour lesquelles il est le plus adapté. Ces travaux montrent une comparaison instructive du compromis entre expressivité et complexité de chaque langage de contraintes. La PPC devrait bénéficier de ce résultat.

Nous donnons ici une définition généralisée de la consistance et de l'opérateur de coupe, puis nous proposons une généralisation des domaines en PPC en domaines abstraits.

2.1 Consistance

Soit E un sous-ensemble de $\mathcal{P}(\mathcal{D})$, avec l'inclusion comme ordre partiel. On notera E^f pour $E\backslash\emptyset$. Par la suite, nous restreindrons les définitions aux cas où E est clos par intersection, car cela est suffisant pour les cas classiques.

Exemple Notons $\mathbb{F}$ l'ensemble des nombres flottants suivant la norme IEEE [7]. Soient $a, b \in \mathbb{F}$, on peut définir $[a, b] = \{x \in \mathbb{R}, a \leq x \leq b\}$ l'intervalle réel délimité par a et b, et $\mathbb{I} = \{[a, b], a, b \in \mathbb{F}\}$ l'ensemble de tous les intervalles à bornes flottantes. $(\mathbb{I}^n, \subset)$ est clos par intersection et forme un treillis. Comme montré par la suite, ceci est l'ensemble de base pour la résolution de contraintes continues.

Définition (et proposition) Soit C une contrainte. Un élément e est E-consistant pour C ssi il est le plus petit élément pour $\mathbf{C_{E,C}} = \{\mathbf{e'} \in \mathbf{E}, \mathcal{S}_\mathbf{C} \subset \mathbf{e'}\}$. Si E est clos par intersection, $\mathbf{C_{E,C}}$ est un treillis complet et il existe un unique élément E-consistant pour C dans E. S'il existe, cet élément est noté $\mathcal{C}_{E,C}$.

Preuve Soit $\mathcal{C}_{E,C} = \cap_{e \subset E, \mathcal{S}_C \subset e} e$. L'existence et l'unicité se déduisent directement du fait que E soit clos par intersection.

Définition Soit $e \in E$, e est E-consistant pour $C_1...C_p$ (ou un sous-ensemble) ssi c'est le plus petit élément de $\{e' \in E, \mathcal{S}_{E,C_1 \wedge ... \wedge C_p} \subset e'\}$. S'il existe, il est noté $\mathcal{C}_{E,C_1 \wedge ... \wedge C_p}$ ou $\mathcal{C}_E$ s'il n'y a pas d'ambiguïtés.

Proposition 2.1 *Si E est clos par intersection, alors c'est un treillis complet et $\mathcal{C}_E$ existe et est unique. De plus, l'ensemble de tous les $\mathcal{C}_{E,C_{i_1} \wedge ... \wedge C_{i_k}}$ pour $i_1...i_k \in \{1...p\}$ forme un treillis par inclusion et $\mathcal{C}_{E,C_1...C_p}$ est le plus petit élément.*

Preuve Soient $i, j \in \{1..p\}$, $\mathcal{C}_{E,C_i}$, $\mathcal{C}_{E,C_j}$ admet un plus petit élément $\mathcal{C}_{E,C_i \wedge C_j} \subset \mathcal{C}_{E,C_i} \cap \mathcal{C}_{E,C_j}$ et un plus grand élément $\cap_{e \in E, \mathcal{C}_{E,C_i} \subset e, \mathcal{C}_{E,C_j} \subset e} e$.

La proposition est plutôt formelle, mais elle assure que, si chaque contrainte d'un CSP vient avec un propagateur, il suffit d'appliquer ces propagateurs de manière récursive, peu importe l'ordre, jusqu'à ce que le point fixe soit atteint, pour obtenir la consistance de ce CSP. Une proposition similaire a déjà été définie pour les consistances existantes dans [1] ou [2].

Nous montrons ci-dessous que les principales consistances existantes, discrètes ou continues, sont incluses dans la définition.

Proposition 2.2 *Soit $S(\mathbb{N}) = \mathcal{P}(\mathbb{N})^n$. La $S(\mathbb{N})$-consistance correspond à la consistance d'arc généralisée (GAC).*

Preuve Nous prouvons d'abord que GAC$\Longrightarrow S(\mathbb{N})$-consistant. Soit $D = D_1 \times ... \times D_n$ GAC pour une contrainte C. D contient évidemment toutes les solutions, il reste donc à prouver que c'est le plus petit élément de $S(\mathbb{N})$. Soit $D' \in S(\mathbb{N})$ strictement plus petit que D, il existe donc un i tel que $D'_i \subsetneq D_i$ et un $v \in D_i \backslash D'_i$. Puisque $v \in D_i$ et D est GAC, il existe aussi $v^j \in D_j$ pour $j \neq i$ tel que $(v^1...v...v^n)$ soit une solution. Cette solution n'est pas dans D' et donc $D' \subsetneq D$ perd des solutions.
Nous prouvons maintenant que $S(\mathbb{N})$-consistant $\Longrightarrow$ GAC. Soit $D = D_1 \times ... \times D_n$ $S(\mathbb{N})$-consistant pour une contrainte C. Soient $i \in \{1...n\}$ et $v \in D_i$. Supposons par l'absurde que pour chaque $v^j \in D_j$, $(v^1...v...v^n)$ ne soit pas une solution : on peut construire l'ensemble $D_1 \times ... \times (D_i \backslash \{v\}) \times ...D_n$ strictement plus petit que D contenant toutes les solutions. Donc D n'est pas le plus petit élément. Il ne peut donc pas exister de tels i et v, et D est GAC.

Proposition 2.3 *Soit $\mathcal{I}(\mathbb{N}) = \{\{\underline{d_1}...\overline{d_1}\} \times ... \times \{\underline{d_n}...\overline{d_n}\}, \underline{d_i}, \overline{d_i} \in \mathbb{N}\}$. La $\mathcal{I}(\mathbb{N})$-consistance est la consistance de bornes.*

Proposition 2.4 *La $\mathbb{I}^n$-consistance est la hull-consistance.*

La définition de consistance abstraite généralise ainsi les consistances existantes. Pour tout $E \subset \mathcal{P}(D)$ clos par intersection, la consistance est bien définie.

2.2 Opérateur de coupe

La consistance n'étant pas suffisante pour résoudre un CSP, il faut aussi un opérateur de coupe.

Définition Soit $(E, \subset)$ un poset. Un *opérateur de coupe* est un opérateur $\oplus : E \to \mathcal{P}(E)$ tel que $\forall e \in E$,
- $|\oplus(e)|$ est fini, $\oplus(e) = \{e_1...e_k\}$,
- $\cup_{1 \leq j \leq k} e_j = e$,
- $\forall j \in \{1...k\}$, $e \neq \emptyset \Longrightarrow e_j \neq \emptyset$,
- $e_j = e \Longrightarrow e$ est le plus petit élément de E^f.

La première condition est nécessaire pour que la résolution termine (largeur de l'arbre de recherche finie). La deuxième condition assure que la coupe ne perd pas de solutions. $\oplus(e)$ peut être une partition de e mais ceci n'est pas nécessaire pour prouver la complétude de la résolution. La troisième condition est purement technique, elle permet de garder l'ensemble vide pour caractériser l'inconsistance. La quatrième condition assure que l'opérateur de coupe coupe effectivement : il est interdit de ne pas couper.

Remarque Il est important de noter que la définition implique : si e n'est pas le plus petit élément de E^f, $e_k \subsetneq e$.

Nous montrons ci-dessous que l'instanciation discrète et la bissection en continu sont incluses dans la définition. Nous utiliserons les notations suivantes pour les domaines abstraits cartésiens : soit $\oplus_1 : E_1 \to \mathcal{P}(E_1)$ un opérateur pour un domaine abstrait E_1. Soient E_2 un autre domaine abstrait et Id la fonction identité pour E_2. On note $\oplus_1 \times Id$ l'opérateur sur $E_1 \times E_2$ tel que $\oplus_1 \times Id(e_1, e_2) = \cup_{e \in \oplus_1(e_1)} e \times e_2$. On notera aussi Id^i pour le produit cartésien de i fois Id.

Exemple L'instanciation d'une variable discrète est un opérateur de coupe sur $\mathcal{P}(\mathbb{N})$: $\oplus_{\mathcal{P}(\mathbb{N})}(d) = \cup_{v \in d}\{v\}$. Pour chaque $i \in \{1...n\}$, l'opérateur $\oplus_{S(\mathbb{N}),i}(d) = Id^{i-1} \times \oplus_{\mathcal{P}(N)} \times Id^{n-i-1}$, avec $\oplus_{\mathcal{P}(N)}$, à la $i$ème place, un opérateur de coupe. Cela revient à choisir une variable V_i et une valeur v dans D_i.

Exemple La même construction permet de définir la coupe en continu en prenant comme opérateur de base sur $\mathbb{I}$: $\oplus_{\mathbb{I}}(I) = \{I^{\vdash}, I^{\dashv}\}$, avec $I^{\vdash}$ et $I^{\dashv}$ deux sous intervalles non vides de I avec n'importe quel moyen de gérer les erreurs d'arrondi sur les flottants, sous réserve que $I^{\vdash} \cup I^{\dashv} = I$.

2.3 Domaines abstraits

Définition Un *domaine abstrait* $D^\sharp$ est défini par :
- un treillis complet E,
- une séquence d'opérateurs de coupe sur E,
- une concrétisation $\gamma : D^\sharp \to D^\flat$, et une abstraction $\alpha : D^\flat \to D^\sharp$ formant une connexion de Gallois *avec un domaine non-relationnel*,
- une fonction strictement décroissante $\tau : D^\sharp \to \mathbb{R}$.

La connexion de Gallois relie le domaine abstrait à une représentation représentable en machine des domaines du CSP. La fonction τ correspond à une mesure du domaine abstrait. C'est un artifice technique pour exprimer la précision du domaine abstrait utilisé pour la terminaison de la résolution.

Les domaines abstraits peuvent être définis indépendamment des domaines d'un CSP. Ils sont destinés à déterminer la forme de la représentation des domaines. Ils peuvent être cartésiens, mais cela n'est plus obligatoire. Notons que nous n'avons pas défini les propagateurs, en effet ils ne peuvent être défini génériquement car ils dépendent des contraintes et de la forme du domaine abstrait choisi. Ils doivent être *ad hoc*.

Il est maintenant possible de définir un solveur abstrait comme une alternance de propagations et de coupes, voir figure 1. Notons que le choix de l'opérateur de coupe à une certaine profondeur doit être stocké afin de garder le solveur cohérent.

Proposition 2.5 *Si E est clos par intersection (H1), n'a pas de chaine infinie décroissante (H2) et si $r \in \tau(E^f)$, alors l'algorithme de résolution de la figure 1 termine et est complet.*

La preuve est assez technique, nous ne donnerons ici que l'intuition. Supposons que l'arbre de recherche soit infini. Par la définition de l'opérateur de coupe, sa largeur est finie, cela signifie qu'il existe une branche infinie. Le long de cette branche, les domaines abstraits sont strictement décroissants tant que e n'est pas le plus petit élément de E^f. L'algorithme converge donc vers un élément e^∞. Si $e^\infty = \emptyset$, l'algorithme s'arrête (cas de l'échec). Si $e^\infty \neq \emptyset$, comme $r > \tau(E^f)$, $r > \tau(e^\infty)$ et l'algorithme termine (cas du succès). La complétude du solveur est directement prouvée grâce à la définition des opérateurs de coupe.

Cette proposition définit un solveur générique pour n'importe quel domaine abstrait E, à condition que les hypothèses (H1) et (H2) soient vérifiées. L'efficacité de ce solveur dépend bien sûr des algorithmes de consistance de E.

Exemple Soit n fixé, l'ensemble $S(\mathbb{N})$ avec l'opérateur de coupe $\oplus_{S(\mathbb{N}),i}$ pour $i \in \{1...n\}$ est un domaine abstrait vérifiant (H1) et (H2). Afin de modéliser le fait que la résolution termine quand toutes les variables sont instanciées, on peut prendre $\tau_1(e) = max(|D_i|, i \in \{1...n\})$ et $r = 1$. Comme vu précédemment, la $S(\mathbb{N})$-consistance est GAC.

int $j \leftarrow 0$ */*profondeur de l'arbre de recherche*/*
int op[] */*à une profondeur j, stocke l'index de l'opérateur de coupe choisi, initialisé uniformément à 0*/*
int width[] */*à une profondeur j, stocke la largeur de l'arbre déjà exploré, initialisé uniformément à 0*/*
abstract domain $e \in E$
$e = \alpha(D)$ */*initialisation aux domaines concrets*/*
répéter
  répéter
    choisir un opérateur de coupe $\oplus_{E,i}$, op[j]$\leftarrow i$
    $e \leftarrow \oplus_{E,i}(e)_{\text{width[j]}}$ */*point backtrackable*/*
    $e \leftarrow E$-consistance(e)
    j++
  jusqu'à e est vide **ou** $\tau(e) \leq r$
  si e est vide **alors**
    width[j]++
  fin si
jusqu'à width[j] $>$ | $\oplus_{\text{op}[j],E}$ | **ou** $\tau(e) \leq r$
si $\tau(e) \leq r$ **alors**
  retourner $\gamma(e)$ */*concrétisation de e*/*
sinon
  retourner échec
fin si

FIGURE 1 – Résolution avec des domaines abstraits. L'ensemble des opérateur de coupe est $\{\oplus_{E,i}, 1 \leq i \leq k\}$. L'exécution s'arrête quand la précision r est atteinte. Chaque fois qu'un opérateur de coupe est choisi, la partition obtenue par l'application de cet opérateur est marquée comme backtrackable.

Exemple Soit n fixé, l'ensemble $\mathbb{I}^n$ avec l'opérateur de coupe $\oplus_{\mathbb{I}^n,i}$ pour $i \in \{1...n\}$ est un domaine abstrait vérifiant (H1) et (H2). Afin de modéliser le fait que la résolution termine quand une certaine précision r est atteinte, on peut prendre $\tau_3(e) = \max(\overline{d_i} - \underline{d_i}$, pour $D_i = [\underline{d_i}, \overline{d_i}], i \in \{1...n\})$ et s'arrêter quand $\tau_3 \leq r$. La résolution sur $\mathbb{I}^n$ correspond à celle utilisée généralement dans les solveurs continus avec la Hull-consistance.

Ces deux exemples montrent comment retrouver les domaines abstraits de bases, qui sont cartésiens.

Il est possible de construire de nouveaux domaines abstraits. Nous détaillerons dans la prochaine section les octogones, la nouveauté est que ce domaine n'est pas cartésien (relationnel).

3 Octogones

Les octogones ont été introduits par Antoine Miné [9] et sont implantés dans Apron [1]. Nous présentons ici une façon de les adapter à la PPC en définissant une consistance, un opérateur de coupe et une précision octogonale.

Définition Soit $\mathcal{V} = (V_1 \dots V_n)$ un ensemble de variables, on appelle *contrainte potentielle* les contraintes de la forme $V_i - V_j \leq c$ avec c une constante.
On appelle *contrainte octogonale* les contraintes de la forme $\pm V_i \pm V_j \leq c$ avec c une constante.

Remarque Les contraintes potentielles et les contraintes d'intervalle ($V_i \geq a$, $V_i \leq b$) sont des cas particuliers des contraintes octogonales.

Définition Un octogone Oct est un ensemble de points satisfaisant une conjonction de contraintes octogonales.

Différentes représentations peuvent être utilisées pour les octogones, nous en présenterons trois, la première dérivant de la définition donnée par Miné. Nous proposons une deuxième représentation qui permet d'utiliser naturellement les techniques de contraintes sur les intervalles. Quant à la troisième, elle est utilisée et décrite par Miné. Ces trois représentations sont illustrées figure 2.

Octogone comme ensemble de contraintes octogonales Plutôt que de stocker tous les points satisfaisants une conjonction de contraintes, un octogone peut être représenté par l'ensemble des contraintes octogonales qu'il satisfait. Notons qu'un octogone a au plus $2n^2$ côtés en dimension n et donc qu'il y a au plus $2n^2$ contraintes octogonales non redondantes le décrivant en dimension n. Il est facile de déterminer si une contrainte octogonale est redondante par rapport à une autre, en effet si il existe deux contraintes avec la même partie gauche on conservera celle avec la plus petite partie droite (par exemple, $V_i - V_j \leq 10$ est redondante par rapport à $V_i - V_j \leq 2$). Un exemple de cette représentation est donné figure 2(a).

Octogone comme intersection de plusieurs boites La forme particulière des contraintes octogonales permet de représenter les octogones comme l'intersection de plusieurs boites, celle de la base d'origine de vecteurs $u_1 ... u_n$, qu'on appellera canonique notée B_{Can}, et celle de bases tournées. À chaque couple de vecteurs de base u_i, u_j avec $i < j$, correspond une base tournée notée $B^{i,j}_{Rot}$.

Définition La base $B^{i,j}_{Rot}$ est obtenue par une rotation d'angle $\alpha = \pi/4$ des deux vecteurs de base u_i, u_j.

Remarque Cette rotation est effectuée en remplaçant symboliquement dans chaque contrainte C_k la

1. http ://apron.cri.ensmp.fr/

variable V_i par $\cos(\alpha)V_i^{i,j} - \sin(\alpha)V_j^{i,j}$ et V_j par $\sin(\alpha)V_i^{i,j} + \cos(\alpha)V_j^{i,j}$, avec $V_i^{i,j}$ et $V_j^{i,j}$ les nouvelles coordonnées de $B_{Rot}^{i,j}$ correspondant aux variables V_i, V_j. On notera $C_k^{i,j}$ cette nouvelle contrainte.

Cette représentation revient à transformer un CSP initial en un autre CSP contenant :
- les variables initiales $(V_1...V_n)$;
- les variables tournées $(V_1^{1,2}, V_2^{1,2}, V_1^{1,3}...V_n^{n-1,n})$, où $V_i^{i,j}$ est la i-ème variable de la base $B_{Rot}^{i,j}$;
- les domaines initiaux $(D_1...D_n)$;
- les domaines tournés $(D_1^{1,2}, D_2^{1,2}, D_1^{1,3}...D_n^{n-1,n})$;
- les contraintes initiales $(C_1...C_p)$;
- les contraintes tournées $(C_1^{1,2}...C_1^{n-1,n}...C_p^{n-1,n})$;

Notons que pour $i = j$, $V_l^{i,j} = V_l$, $D_l^{i,j} = D_l$ et $C_k^{i,j} = C_k$ avec $i, j, l \in \{1...n\}$ et $k \in \{1...p\}$. De même, pour $l \neq i$ et $l \neq j$, $V_l^{i,j} = V_l$ et $D_l^{i,j} = D_l$.

Par la suite on notera $V_1...V_n^{n-1,n}$ l'ensemble des variables de toutes les bases prenant leurs valeurs dans les domaines $D_1...D_n^{n-1,n}$ et $C_1...C_p^{n-1,n}$ l'ensemble de toutes les contraintes.

Un exemple en dimension 2 est donné figure 2(b).

Octogone avec une DBM Un octogone peut être stocké à l'aide d'un matrice à différences bornées (DBM) comme proposé par Miné.

Définition Une DBM est une matrice carrée $n \times n$. L'élément à la ligne i et colonne j, avec $i, j \in \{1 \dots n\}$, est égal à c s'il existe une contrainte $V_j - V_i \leq c$, et $+\infty$ sinon.

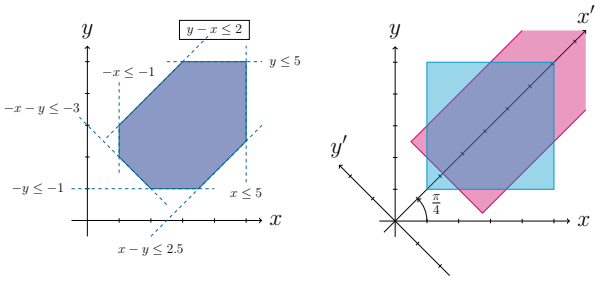
Afin de représenter un octogone avec une DBM, il est nécessaire de réécrire les contraintes octogonales en contraintes potentielles.

Soit $\mathcal{V}' = (V'_1 \dots V'_{2n})$ un ensemble contenant deux fois plus de variables. Chaque variable V_i a une *forme positive* $V'_{2i-1} = V_i$, et une *forme négative* $V'_{2i} = -V_i$. Ce nouvel ensemble permet de transformer les contraintes octogonales dans $\mathcal{V}$ en contraintes potentielles dans $\mathcal{V}'$.

Exemple La contrainte $V_i + V_j \leq c$ se réécrit $V'_{2i-1} - V'_{2j} \leq c$ et $V'_{2j-1} - V'_{2i} \leq c$.

Grâce à cette réécriture, un octogone peut être représenté par un DBM de dimension $2n$. Par exemple, figure 2(c), l'élément à la ligne x et colonne y correspond à la contrainte $y - x \leq 2$.

Nous avons vu, section 2, que pour utiliser un domaine abstrait en PPC, il est nécessaire de définir une consistance, une séquence d'opérateurs de coupe, des connexions de Gallois et une fonction de précision.



(a) (b)

	x	$-x$	y	$-y$
x	0	−2	**2**	−3
$-x$	10	0	$+\infty$	2.5
y	2.5	−3	0	−2
$-y$	$+\infty$	**2**	10	0

(c)

FIGURE 2 – Différentes représentations d'un octogone : représentation géométrique avec les contraintes octogonales 2(a), comme l'intersection de deux boites 2(b) et la DBM 2(c).

3.1 Consistance

Notre but étant d'utiliser une représentation différente pour les domaines, il est nécessaire de redéfinir un certains nombres d'outils essentiels à la résolution. Dans un premier temps nous nous intéresserons à la consistance.

Un ensemble est dit consistant pour un ensemble de contraintes s'il est le plus petit élément contenant l'ensemble des solutions $\mathcal{S}$. Calculer un octogone consistant revient donc à calculer le plus petit octogone contenant les solutions.

Définition En utilisant le CSP transformé, la consistance octogonale peut se définir comme la hull-consistance sur l'ensemble des domaines $D_1...D_n^{n-1,n}$.

En pratique cette définition de la consistance est trop coûteuse en terme de temps de calcul, dû à la multiplication du nombre de variables et de contraintes à propager. En effet, en plus des contraintes de chacune des bases, il faut ajouter les contraintes de *channeling* permettant de propager l'information entre les différentes bases. On définira donc la consistance octogonale sur l'intersection de boites.

Définition Soit C une contrainte, et $i, j \in \{1...n\}$. Notons $\mathcal{B}^{i,j}$ la Hull-consistant boite pour la contrainte tournée $C^{i,j}$, et $\mathcal{B}$ la Hull-consistant boite pour C. L'octogone consistant pour C est l'intersection de toutes les $\mathcal{B}^{i,j}$ et de $\mathcal{B}$.

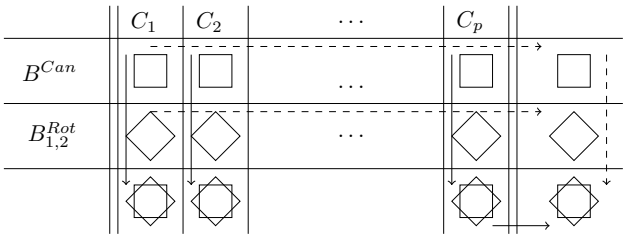


FIGURE 3 – Schéma des possibles heuristiques pour la consistance octogonale. En pointillés la première version, la propagation des deux CSPs est faite en parallèle. En trait plein la seconde version, la consistance octogonale est calculée pour chaque contrainte.

Comme montré sur la figure 3, on peut imaginer deux stratégies. La première, en pointillés, applique la consistance en parallèle dans chaque base, une seule passe sur les contraintes est effectuée, puis récupère l'octogone par intersection des boites obtenues. Étant donné que cette intersection peut réduire les boites, ces deux étapes sont répétées jusqu'à l'obtention du point fixe. N'effectuer qu'une seule passe sur les contraintes d'une base permet d'éviter une convergence lente de la consistance de cette base.

Dans la seconde version, en trait plein sur la figure 3, on calcule la consistance pour une contrainte donnée dans chacune des bases. L'octogone consistant pour cette contrainte est calculé par l'intersection des boites obtenues et est le point de départ pour la prochaine contrainte à propager. Ceci est répété jusqu'à l'obtention du point fixe. Cette seconde version correspond à la version intuitive de la consistance octogonale.

Remarque Quelque soit la version choisie, l'octogone consistant est le même à la fin. Cette représentation des octogones est close par intersection, la preuve est donc directe grâce à la Proposition 2.1.

Il reste un problème, comment réaliser l'intersection de boites ne se trouvant pas dans la même base. Pour palier à ce problème nous utilisons une DBM pour représenter l'octogone, ainsi l'intersection se fait en exécutant une version modifiée de l'algorithme de plus court chemin de Floyd-Warshall sur cette matrice. Chercher le plus court chemin entre V_i et V_j revient à chercher la plus petite constante c pour la contrainte $V_i - V_j \leq c$. En plus des chemins passant par un autre sommet V_k, cette version modifiée prend en compte des chemins supplémentaires dû au doublement des variables dans la DBM. Chaque variable apparait deux fois, sous forme positive et sous forme négative. On sait que $V_i - V_j \leq c$ est équivalent à $2V_i - 2V_j \leq 2c$, or si l'on représente la DBM sous forme de graphe, il n'existe pas de chemins pour la seconde inégalité. Cette absence de

```
Octogone oct
boolean modif /*vrai si domaine est réduit, faux sinon*/
float M[2n][2n] /*DBM associée à oct*/
pour chaque base b de oct faire
  b ← Hull-consistance(b)
  si il y a eu une réduction alors
    Modifier les cases correspondantes de M
    modif ← true
  fin si
fin pour
tant que modif faire
  modif ← false
  M ← Floyd-Warshall-Modif(M)
  pour chaque cellule qui a été modifiée par Floyd-
  Warshall-Modif faire
    Changer le domaine de la variable correspondante
  fin pour
  pour chaque base b dont au moins un des domaines
  a été réduit par Floyd-Warshall-Modif faire
    b ← Hull-consistance(b)
    si il y a eu une réduction alors
      Modifier les cases correspondantes de M
      modif ← true
    fin si
  fin pour
fin tant que
retourner oct
```

FIGURE 4 – Pseudo code de la première version de la consistance octogonale.

chemins rend la version originale de Floyd-Warshall incomplète, d'où la nécessité d'une version modifiée. Comme prouvé dans [9], cet algorithme réduit les domaines des différentes boites sans modifier l'octogone initial. Son exécution sur la DBM permet de simuler la propagation des contraintes octogonales.

La consistance ainsi définie suppose de maintenir les deux représentations différentes des octogones mais permet en pratique un gain de temps.

La figure 4 présente le pseudo-code de la première version de la consistance octogonale. Cet algorithme termine, est correct et est confluent, quel que soit l'ordre dans lequel les consistances sont appliquées l'octogone consistant est le même.

Nous avons maintenant les outils nécessaires à l'approximation extérieure d'un ensemble de solutions à l'aide d'un octogone. Nous n'avons cependant pas encore les éléments essentiels à la résolution d'un CSP à l'aide d'octogones. En particulier, il reste à définir un opérateur de coupe et une précision octogonale.

3.2 Résolution

En continu, les techniques de résolution choisissent généralement un domaine, le coupent en deux (ou plus)

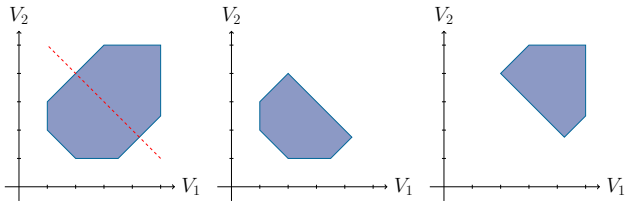


FIGURE 5 – Exemple d'une coupe : l'octogone à gauche est coupé dans la base $B^{1,2}$.

et appliquent une consistance sur les deux boites obtenues. Ces étapes sont répétées tant qu'une certaine précision n'est pas atteinte. Dans le cas de la résolution avec des octogones, il nous faut donc définir les trois points clés de la résolution qui sont l'opérateur de choix du domaine à couper, l'opérateur de coupe et la précision.

Opérateur de choix La représentation d'un octogone comme l'intersection de boites permet ici de définir directement l'opérateur de choix, noté $\mathbb{O}_{Oct}(D_1...D_n, D_1^{1,2}...D_n^{n-1,n})$. En effet, avec cette représentation, les opérateurs de choix usuels sont utilisables. En pratique on choisira le plus grand domaine comme prochain domaine à couper. Deux stratégies sont possibles, la première, est de restreindre le choix aux domaines des variables de la base canonique. La seconde, est d'étendre le choix aux domaines des variables de toutes les bases.

Définition L'opérateur de choix peut être défini comme suit : $\mathbb{O}_{Oct}(D_1...D_n^{n-1,n}) = D_i^{i,j}$ avec $i,j \in \{1...n\}$ et $\overline{d_i^{i,j}} - \underline{d_i^{i,j}} = max(\overline{d_k^{k,l}} - \underline{d_k^{k,l}}$, pour $D_k^{k,l} = [\underline{d_k^{k,l}}, \overline{d_k^{k,l}}], k,l \in \{1...n\})$.

Dans le cas de la première stratégie, on pose $i = j$.

Opérateur de coupe Une fois le domaine à couper choisi, il faut déterminer comment le couper. Toujours en utilisant la représentation de l'intersection des boites, l'opérateur de coupe octogonal peut être défini comme suit :

Définition (et proposition) $\oplus_{Oct}(D_1...D_n^{n-1,n}) = \{D_1...D_i^{i,j\vdash}...D_n^{n-1,n}, D_1...D_i^{i,j\dashv}...D_n^{n-1,n}\}$, avec $D_i^{i,j\vdash}$ et $D_i^{i,j\dashv}$ deux parties non vides de D_i, $D_i^{i,j\vdash} \cup D_i^{i,j\dashv} = D_i$ et $D_i = \mathbb{O}_{Oct}(D_1...D_n^{n-1,n})$.

Ainsi défini, l'opérateur de coupe ne prend pas en compte les corrélations entre les variables des bases. Il faut donc ajouter une contrainte dans les autres bases afin de communiquer cette modification aux autres bases.

```
Octogone oct /*octogone initial*/
queue splittingList ← oct /*liste d'octogones à couper*/
queue acceptedOct ← ∅ /*liste d'octogones acceptés*/
tant que splittingList ≠ ∅ faire
  Octogone octAux ← splittingList.top()
  splittingList.pop()
  octAux ← Oct-consistance(octAux)
  si τ(octAux) < r ou octAux ne contient que des solutions alors
    Ajouter octAux à acceptedOct
  sinon
    Octogone octGauche ← gauche(⊕_Oct(octAux))
    Octogone octDroit ← droit(⊕_Oct(octAux))
    splittingList.add(octGauche)
    splittingList.add(octDroit)
  fin si
fin tant que
retourner acceptedOct
```

FIGURE 6 – Résolution avec des octogones.

Exemple Soit $D_i = \mathbb{O}_{Oct}(D_1...D_n^{n-1,n}), i \in \{1...n\}$ le domaine choisi par l'opérateur de coupe. Il faut ajouter dans toutes les bases obtenues par rotation des vecteurs de bases u_i, u_j la contrainte :

$$\cos(\alpha)V_i^{i,j} - \sin(\alpha)V_j^{i,j} \diamond c \text{ si } i < j$$
$$\sin(\alpha)V_i^{i,j} + \cos(\alpha)V_j^{i,j} \diamond c \text{ si } i > j$$

où $j \in \{1...n\}, \diamond \in \{\leq, \geq\}$ (suivant le côté) et c est la valeur sur laquelle le domaine a été coupé.

Un exemple de coupe est présenté figure 5.

De même que les opérateurs de choix et de coupe, il faut définir ce que signifie la précision octogonale.

Précision Dans la plupart des solveurs continus, la précision est définie comme la taille du plus grand domaine. Pour les octogones, cette définition ne peut être utilisée car elle repose uniquement sur la taille des domaines et ne prend pas en compte la corrélation entre les variables.

Définition (et proposition) Soit $\mathcal{O}$ un octogone, on définit la précision octogonale $\tau(\mathcal{O}) = \min_{1\leq i,j\leq n}(\max_{1\leq k\leq n}(\overline{d_k^{i,j}} - \underline{d_k^{i,j}}))$.

Pour une seule boite, la définition de τ est identique à la précision usuelle. Pour un octogone, nous prenons le minimum des précisions des boites de toutes les bases. Même si techniquement la définition est différente, le sens de la précision octogonale est identique à celui de la précision usuelle : dans un octogone de précision r, tout point est à distance au plus r de l'ensemble des solutions $\mathcal{S}$.

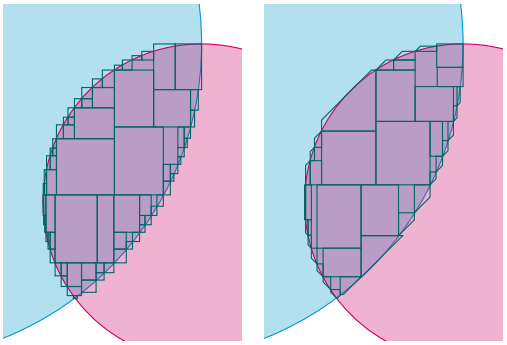


FIGURE 7 – Les résultats obtenus avec une limite de temps (7ms). à gauche la résolution dans $\mathbb{I}^n$ de IBEX. à droite, la résolution octogonale avant la concrétisation.

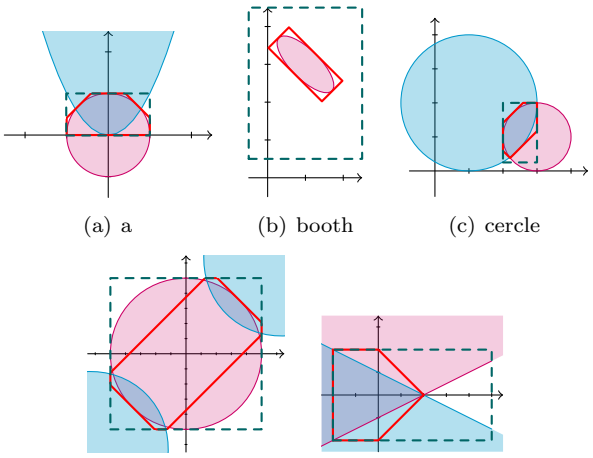


FIGURE 8 – Exemples de problèmes en dimension 2, avec la consistance par intervalles en pointillés vert et la consistance octogonale en rouge.

Résolution La figure 6 décrit l'algorithme de résolution octogonale. Cet algorithme termine et est correct. Il retourne la liste des octogones solutions, c'est-à-dire ceux ne contenant que des solutions ou dont la précision est inférieure à un paramètre r. Notons que dans l'hypothèse où l'on souhaite renvoyer une représentation non relationnelle, la concrétisation peut être faite en utilisant la définition donnée par Miné [9].

4 Résultats

Un prototype de résolution de problèmes continus à l'aide d'octogones comme représentation a été implanté dans IBEX, un solveur continu [4]. Dans cette implantation la consistance octogonale utilise la procédure *HC4-Revise* [3] pour contracter les contraintes de chacune des bases. Les deux versions de consistance octogonale décrites précédemment ont été implantées, et en pratique leur temps d'exécution est équivalent. De même les deux stratégies pour l'opérateur de choix ont été implantées. Le paramètre de précision est fixé à 0.001 ($r = 0.001$). Après la création d'un octogone, les contraintes sont simplifiées afin de réduire les multiples occurrences des variables. Pour cette étape la fonction *Simplify* de Mathematica est utilisée.

La figure 7 compare les résultats obtenus, sur un problème simple d'intersection de deux disques, par d'un côté la résolution usuelle avec les intervalles et de l'autre celle avec les octogones. Pour cette expérimentation, la précision n'est pas fixée et nous utilisons une limite de temps, de 7ms, pour arrêter l'algorithme de résolution.

Le tableau 1 compare le temps CPU, le nombre de nœuds solutions et le nombre de points de choix obtenus par la résolution par intervalles avec *HC4* à ceux obtenus par la résolution par octogones. Deux versions de la résolution octogonale sont présentées, dans la première, Oct-v1, l'opérateur de choix choisit parmi tous les domaines (de la base canonique et des bases tournées), alors que dans la seconde, Oct-v2, seuls les domaines de la base canonique peuvent être choisis.

Ce tableau montre les bons résultats obtenus par la résolution octogonale, en termes de pouvoir de filtrage (petit nombre de points de choix) et de contraction (petit nombre d'octogones solutions). De plus, on constate que la phase de simplification des contraintes améliore grandement les résultats, le temps CPU est réduit grâce à un meilleur filtrage et une meilleure contraction. Notons que les temps n'incluent pas la simplification de Mathematica. En pratique ce temps est négligeable.

La figure 8 compare sur les problèmes testés, la consistance obtenue avec l'algorithme HC4 sur les intervalles et la consistance octogonale. Pour tous ces exemples, la consistance octogonale est plus précise que la consistance usuelle en continu. Notons que HC4 calcule la hull-consistance sur l'ensemble des contraintes primitives sans les décomposer. Cette forme compacte peut générer une surapproximation notamment lors d'occurrences multiples des variables.

Sur le premier exemple, 8(a), la consistance octogonale ne réduit pas autant qu'espéré, cela est dû aux nouvelles multiples occurrences des variables qui apparaissent dans la contrainte parabolique dans la base tournée. Au contraire, sur le deuxième exemple 8(b), il existe plusieurs occurrences des variables dans la base canonique qui disparaissent dans la base tournée.

		Avec Simplifications			Sans Simplifications	
	$\mathbb{I}^n$	Oct-v1		Oct-v2	Oct-v1	Oct-v2
a	0.31 *4905* 9809	0.57 *1444* 3175	0.59 *1742* 3483		1.39 *2347* 5171	1.17 *2462* 4923
booth	0.42 *8773* 19421	0.28 *956* 2005	0.37 *1116* 2431		3.18 *4972* 11863	2.64 *5163* 10865
cercle	0.37 *5337* 10673	0.32 *897* 1793	0.32 *880* 1759		1.34 *2330* 4939	1.16 *2399* 4797
hs8	1.2 *18596* 37191	0.7 *2225* 4449	0.82 *2215* 4429		6.36 *10299* 22349	5.2 *10871* 21741
triangle	0.77 *11897* 23793	0.63 *1967* 3933	0.74 *1903* 3805		3.48 *6253* 12529	3.1 *6537* 13073

TABLE 1 – Résultats expérimentaux. Chaque colonne donne le temps CPU en secondes (à gauche), et nombre de point de choix (en bas à droite) et le nombre de nœuds solutions (en haut à droite).

5 Conclusion

Cet article a présenté une nouvelle représentation des domaines s'inspirant des domaines abstraits de l'Interprétation Abstraite. Il définit les outils nécessaires pour résoudre avec des domaines abstraits qui généralisent les domaines habituels de la PPC. De plus, il décrit un exemple détaillé, celui des octogones.

Lors de la création de l'octogone, toutes les bases tournées sont générées. On obtient de meilleurs performances sur de petits exemples, mais cette génération systématique risque de ne pas passer à l'échelle. Dans des travaux futurs, une étude syntaxique des contraintes permettrait de ne générer que certaines bases. Une heuristique d'octogonalisation permettrait la création d'un octogone plus adapté au problème à résoudre en choisissant les contraintes à octogonaliser et un angle de rotation plus adéquat.

Ces travaux sont un premier pas vers l'utilisation des domaines abstraits en Programmation par Contraintes. De plus, les définitions génériques des différents points clés de la résolution laissent entrevoir un cadre uniforme pour la résolution de problèmes mixtes discret/continu.

Références

[1] Krzysztof R. Apt. The essence of constraint propagation. *Theoretical Computer Science*, 221, 1999.

[2] Frédéric Benhamou. Heterogeneous constraint solvings. In *Proceedings of the 5th International Conference on Algebraic and Logic Programming*, pages 62–76, 1996.

[3] Frédéric Benhamou, Frédéric Goualard, Laurent Granvilliers, and Jean-François Puget. Revisiting hull and box consistency. In *Proceedings of the 16th International Conference on Logic Programming*, pages 230–244, 1999.

[4] Gilles Chabert and Luc Jaulin. Contractor programming. *Artificial Intelligence*, 173 :1079–1100, 2009.

[5] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *Proceedings of the 6th ACM SIGACT-SIGPLAN symposium of Principles of Programming Languages*, pages 269–282, 1979.

[6] Patrick Cousot and Radia Cousot. Static determination of dynamic properties of programs. In *Proceedings of the 2nd International Symposium on Programming*, pages 106–130, 1976.

[7] David Goldberg. What every computer scientist should know about floating point arithmetic. *ACM Computing Surveys*, 23(1) :5–48, 1991.

[8] Antoine Miné. *Domaines numériques abstraits faiblement relationnels.* PhD thesis, École Normale Supérieure, December 2004.

[9] Antoine Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19(1) :31–100, 2006.

Utilisation de solveurs de contraintes pour réduire les approximations produites par interprétation abstraite

Olivier Ponsini Claude Michel Michel Rueher

Université de Nice–Sophia Antipolis, I3S/CNRS
BP 121, 06903 Sophia Antipolis Cedex, France
prénom.nom@unice.fr

Résumé

Les programmes comportant des calculs avec des nombres flottants sont délicats à concevoir car l'arithmétique flottante diffère de l'arithmétique réelle par de nombreuses propriétés contre-intuitives. Une approche classique pour vérifier de tels programmes consiste à estimer la précision des calculs avec des flottants par rapport à la même séquence d'opérations effectuées avec une sémantique idéale sur les réels. Des outils comme Fluctuat, basés sur l'interprétation abstraite, ont été développés pour traiter cette question. Toutefois, ces outils sont confrontés à un problème de sur-approximation du domaine des variables, aussi bien avec la sémantique des nombres flottants qu'avec la sémantique des nombres réels. Cette sur-approximation peut-être très grossière pour certains programmes. Dans cet article, nous montrons que des solveurs de contraintes sur les flottants et sur les réels peuvent améliorer significativement les approximations calculées par Fluctuat. Nous avons réussi à fortement réduire les domaines des variables de programmes C difficiles pour les techniques d'interprétation abstraite implémentées dans Fluctuat.

Abstract

Programs with floating-point computations are tricky to develop because floating-point arithmetic differs from real arithmetic and has many counterintuitive properties. A classical approach to verify such programs consists in estimating the precision of floating-point computations with respect to the same sequence of operations in an idealized semantics of real numbers. Tools like Fluctuat—based on abstract interpretation—have been designed to address this problem. However, such tools compute an over-approximation of the domains of the variables, both in the semantics of the floating-point numbers and in the semantics of the real numbers. This over-approximation can be very coarse on some programs. In this paper, we show that constraint solvers over floating-point numbers and real numbers can significantly refine the approximations computed by Fluctuat. We managed to reduce drastically the domains of variables of C programs that are difficult to handle for abstract interpretation techniques implemented in Fluctuat.

1 Introduction

De nombreuses applications logicielles critiques emploient des calculs avec des nombres flottants : notamment, les applications de contrôle ou de simulation des systèmes physiques dans différents domaines tels que le transport, le nucléaire, le médical, ou encore l'aéronautique et le spatial. Ces applications sont souvent basées sur des modèles mathématiques et des algorithmes conçus pour les nombres réels. Les nombres flottants sont alors une source supplémentaire possible d'erreurs, comme en témoignent les nombreuses défaillances tristement célèbres dues aux flottants, e.g., l'explosion de la fusée Ariane ou l'échec de l'antimissile Patriot. En effet, pour une même séquence d'opérations, le comportement des flottants n'est pas identique à celui des réels. La nature finie des nombres flottants a des conséquences qui vont à l'encontre de l'intuition acquise sur les réels [12, 21]. Par exemple, certains nombres décimaux d'apparence simples ne sont pas des flottants (e.g., $0,1$ n'a pas de représentation exacte sur les flottants binaires), les opérateurs arithmétiques ne sont pas associatifs et sujets aux phénomènes d'absorption (e.g., $a + b$ est arrondi à a quand a est très supérieur à b) ou d'annulation (soustraction de deux nombres proches après arrondi qui ne

conserve que l'erreur d'arrondi). En dépit du standard IEEE 754, ces phénomènes intrinsèques de l'arithmétique flottante dépendent aussi de nombreux facteurs tels que les compilateurs, les systèmes d'exploitations ou les architectures matérielles.

Malgré la connaissance de ces phénomènes, s'assurer qu'un programme avec des flottants approxime convenablement le modèle mathématique sur les réels dont il est issu reste très difficile. Des outils de vérification formelle ont été développés pour estimer l'exactitude des calculs flottants et pour montrer l'absence de certaines erreurs à l'exécution comme le débordement de capacité, les opérations invalides ou la division par zéro. Les outils automatiques existants sont principalement basés sur les techniques d'interprétation abstraite. En particulier, l'analyseur statique FLUCTUAT [10] a été employé avec succès pour étudier la propagation des erreurs d'arrondi dans des programmes C industriels critiques[1]. FLUCTUAT calcule deux sur-approximations des domaines des variables d'un programme considéré respectivement avec la sémantique des réels et avec la sémantique des flottants. FLUCTUAT est alors capable de borner l'erreur commise avec les flottants. Les sur-approximations calculées sont suffisamment précises pour les besoins de l'analyse dans certains cas. Cependant, des constructions usuelles des programmes conduisent à des sur-approximations trop grandes pour permettre à l'analyse d'être conclusive.

Dans cet article, nous montrons que des solveurs de contraintes sur les flottants et sur les réels peuvent raffiner significativement les approximations calculées par FLUCTUAT. Nous exploitons les capacités de réfutation des algorithmes de filtrage pour réduire les domaines calculés par interprétation abstraite. Notre implémentation utilise des solveurs de contraintes sur intervalles : REALPAVER [15] qui est un solveur correct de contraintes sur les réels et FPCS [20, 19] qui est un solveur correct de contraintes sur les flottants. Nous avons réussi à fortement réduire les domaines des variables de programmes C difficiles pour les techniques d'interprétation abstraite implémentées dans FLUCTUAT.

La section 2 illustre notre approche sur un petit exemple et présente les principaux travaux existants. La section 3 détaille notre approche et les différents outils qui la compose : FLUCTUAT, REALPAVER et FPCS. La section 4 discute les résultats de notre approche sur plusieurs programmes représentatifs.

[1]Les applications critiques concernées par les outils de vérification formelle des calculs flottants sont souvent embarquées et majoritairement écrites en langage C.

```
1 /* Pré−condition :  x ∈ [0 .. 10] */
2 double conditionnelle(double x) {
3   double y = x*x - x;
4
5   if (y >= 0)
6     y = x/10;
7   else
8     y = x*x + 2;
9
10  return y;
11 }
```

FIG. 1 – Intersection de domaines abstraits

2 Motivations

Dans cette section, nous illustrons notre approche par un exemple motivant, puis nous positionnons notre approche par rapport aux travaux existants.

2.1 Exemple

Le programme de la figure 1 est cité dans [11] pour les difficultés qu'il pose aux analyses par interprétation abstraite. Sur les flottants, comme sur les réels, cette fonction renvoie une valeur comprise dans l'intervalle $[0 .. 3]$ alors que l'outil FLUCTUAT calcule l'intervalle $[0..102]$. En effet, nous pouvons déduire de l'instruction conditionnelle de la ligne 5 que :

- branche `si` : $x = 0$ ou $x \geq 1$ et donc $y \in [0 .. 1]$;
- branche `sinon` : $x \in]0 .. 1[$ et donc $y \in [2 .. 3]$.

Cependant, tous les domaines abstraits classiques (e.g., intervalles, polyèdres), ainsi que le domaine des *zonotopes* de FLUCTUAT, échouent à obtenir une bonne approximation de cette valeur. La difficulté pour ces analyses est de réaliser l'intersection des domaines abstraits calculés pour y en lignes 3 et 5. Ces analyses sont notamment incapables d'en déduire une contrainte sur x ; elles considèrent ainsi que dans la branche `sinon` le domaine de x est toujours l'intervalle $[0 .. 10]$.

Dans l'approche que nous proposons, nous calculons une approximation des domaines dans chacun des deux chemins d'exécution. Les techniques de filtrage de la programmation par contraintes sont suffisamment fortes pour réduire les domaines des variables des systèmes de contraintes correspondants générés. Considérons par exemple le système de contraintes sur les réels $\{y_0 = x_0 * x_0 - x_0, y_0 < 0, y_1 = x_0 * x_0 + 2, x_0 \in [0..10]\}$ qui correspond au chemin d'exécution passant par la branche `sinon` dans la fonction `conditionnelle`[2]. À

[2]Les instructions des programmes sont converties en forme SSA (*Static Single Assignment*) dans laquelle chaque variable n'est affectée qu'une seule fois dans chaque chemin d'exécution.

	Domaine	Temps
Exact (réels et flottants)	[0 .. 3]	n.d.
FLUCTUAT (réels et flottants)	[0 .. 102]	0,1 s
Zonotopes contraints (réels)	[0 .. 9,72]	n.d.
FPCS (flottants)	[0 .. 3,027]	0,2 s
REALPAVER (réels)	[0 .. 3,001]	0,3 s

TAB. 1 – Domaine de la fonction `conditionnelle`

partir des contraintes $y_0 = x_0 * x_0 - x_0$ et $y_0 < 0$, un solveur de contraintes sur intervalles peut réduire le domaine initial de x_0 à [0 .. 1]. Ce domaine réduit servira alors à calculer celui de y_1 via la contrainte $y_1 = x_0 * x_0 + 2$, pour produire $y_1 \in [2 .. 3{,}001]$. De la même façon, un solveur de contraintes sur les flottants réduit le domaine de x_0 à $[4{,}94 \cdot 10^{-324} .. 1{,}026]$ et celui de y_1 à [2 .. 3,027].

En résumé, nous construisons les systèmes de contraintes correspondant à chaque chemin exécutable dans une fonction et nous utilisons des techniques de filtrage pour réduire les domaines des variables calculés par FLUCTUAT. Les chemins d'exécution sont explorés à la volée et interrompus aussitôt qu'est détectée l'inconsistance du système de contraintes associé. Sur les nombres réels, nous utilisons la combinaison des consistances de boîte (*box*) et d'enveloppe (*hull*) implémentée dans REALPAVER [15]. Sur les nombres flottants, nous utilisons la $3B$-consistance [18] implémentée dans FPCS [20]. Les résultats des différentes méthodes sur l'exemple de la fonction `conditionnelle` sont donnés dans la table 1. Sur cet exemple, contrairement à FLUCTUAT, notre approche fournit une très bonne approximation, à la fois sur les réels et sur les flottants. Les temps d'analyse sont très similaires (n.d. signifie non disponible). Dans [11], les auteurs proposent une extension aux zonotopes, appelée zonotopes contraints (*constrained zonotopes*), qui tente de pallier le problème dû aux instructions conditionnelles des programmes. Cette extension est définie sur les réels et n'est pas encore implémentée dans FLUCTUAT. L'approximation calculée avec les zonotopes contraints est meilleure que celle de FLUCTUAT, mais elle reste moins précise que celle calculée avec REALPAVER.

2.2 Travaux connexes

Différentes méthodes ont été appliquées au problème de la validation statique des programmes comportant des calculs en nombres flottants : notamment, les analyses par interprétation abstraite et les preuves de programmes avec des assistants de preuve ou avec des procédures de décision dans des solveurs automatiques.

Les méthodes d'analyse par interprétation abstraite représentent les erreurs d'arrondi des calculs en arithmétique flottante dans leurs domaines abstraits. Elles sont en général rapides, automatiques et passent bien à l'échelle. En revanche, elles peuvent manquer de précision et ne génèrent pas de contre-exemple. ASTRÉE [7] est sans doute l'un des outils les plus connus issu de cette famille de méthodes. En estimant la valeur des variables en tout point d'un programme, ASTRÉE peut prouver l'absence d'erreur à l'exécution, au sens de comportement non défini par le langage de programmation (e.g., division par zéro, débordement de capacité). FLUCTUAT, que nous détaillons en section 3.1, calcule en plus une estimation de l'exactitude des calculs, c.-à-d., une borne de la différence entre les valeurs des variables lorsque le programme est interprété avec une sémantique sur les réels et lorsqu'il est interprété avec une sémantique sur les flottants [10].

Un deuxième groupe de méthodes s'attache à formaliser l'arithmétique flottante dans des assistants de preuve tels que Coq [2] ou HOL [16]. Les preuves des propriétés des programmes sont alors effectuées interactivement dans l'assistant, qui garantit la correction des preuves. Ces formalisations ne sont pas adaptées à l'évaluation des domaines des variables des programmes. De plus, lorsque la construction d'une preuve échoue, cela n'implique pas qu'une propriété est fausse. Par conséquent, ces approches ne peuvent fournir aucun contre-exemple. L'outil Gappa [9] combine le calcul d'intervalle et l'application de théorèmes prédéfinis par réécriture. Les théorèmes permettent de réécrire les expressions arithmétiques afin de compenser certaines des faiblesses du calcul d'intervalle, e.g., perte des dépendances entre variables. Lorsque les intervalles calculés ne sont pas assez précis, il est possible d'ajouter manuellement des théorèmes ou de subdiviser les domaines d'entrée. Le coût de cette méthode semi-automatique est alors important. Dans [1], les auteurs proposent une axiomatisation de l'arithmétique flottante dans la logique du premier ordre qui permet d'automatiser les preuves dans un assistant tel que Coq en appelant des solveurs SMT (*Satisfiability Modulo Theories*) et Gappa. Leurs expérimentations montrent qu'une interaction humaine avec l'assistant de preuve demeure nécessaire. Du fait de la taille du domaine des variables flottantes, l'approche classique par vecteurs de bits des solveurs SAT est inopérante. Une technique d'abstraction a été proposée pour CBMC [4]. Cette technique est basée sur la sous et la sur-approximation d'un flottant selon une précision exprimée en nombre de bits de la mantisse. Cette technique reste cependant lente.

3 Approche proposée

L'approche que nous proposons consiste à raffiner les domaines des variables calculés par FLUCTUAT en utilisant des solveurs de contraintes sur les réels et les flottants. Avant d'exposer les détails de notre approche, nous rappelons les caractéristiques de FLUCTUAT (section 3.1), REALPAVER (section 3.2) et FPCS (section 3.3) qui sont utiles à la compréhension de la suite de cet article. L'ensemble du processus de l'approche que nous proposons est décrit en section 3.4.

3.1 Fluctuat

FLUCTUAT [10] est un analyseur statique de programmes C spécialisé dans l'évaluation de la précision des calculs numériques avec des flottants. L'outil compare le comportement du programme analysé sur les réels et sur les flottants. Concrètement, FLUCTUAT permet de spécifier des intervalles de valeurs pour les variables d'entrée d'un programme et calcule pour chaque variable du programme :

- un encadrement du domaine de la variable considérée sur les réels ;
- un encadrement du domaine de la variable considérée sur les flottants ;
- un encadrement de l'erreur maximum entre la valeur réelle et flottante ;
- la contribution de chaque instruction à l'erreur associée à la variable ;
- la contribution des variables d'entrée à l'erreur associée à la variable.

FLUCTUAT procède par interprétation abstraite et utilise le domaine abstrait faiblement relationnel des zonotopes [13], qui établit un bon compromis entre performance et précision. Les zonotopes sont des ensembles de formes affines qui préservent les corrélations linéaires entre variables. Pour améliorer la précision de l'analyse, l'outil permet d'utiliser des nombres en précision arbitraire ou de subdiviser les domaines des variables d'entrée.

FLUCTUAT est développé par le CEA-LIST[3] et a été utilisé avec succès pour des applications industrielles de plusieurs dizaines de milliers de lignes de code dans les domaines du transport, du nucléaire ou de l'aéronautique.

3.2 RealPaver

REALPAVER[4] [15] est un solveur sur intervalles pour des systèmes de contraintes numériques sur les nombres réels. Les contraintes peuvent être non-linéaires et peuvent contenir les opérateurs arithmétiques usuels ainsi que les fonctions transcendantes élémentaires. En revanche, ni les opérateurs d'inégalité stricte, ni les disjonctions de contraintes ne sont disponibles.

Le solveur calcule des approximations fiables d'ensembles continus de solutions en utilisant des méthodes du calcul des intervalles correctement arrondies et des techniques de satisfaction de contraintes. Plus précisément, les domaines calculés sont des intervalles fermés dont les bornes sont des flottants. REALPAVER implémente plusieurs consistances partielles : *box*, *hull* et $3B$ par exemple. L'approximation d'une solution est décrite par une boîte, c.-à-d., le produit cartésien des domaines des variables. Soit REALPAVER prouve qu'un système de contraintes est inconsistant, soit il calcule une union de boîtes qui contient toutes les solutions du système.

3.3 FPCS

FPCS [20, 19] est un solveur de contraintes qui a été conçu pour résoudre correctement, c.-à-d. sans perdre de solutions, un ensemble de contraintes sur les flottants. Pour ce faire, FPCS utilise une $2B$-consistance [18] ainsi que des fonctions de projections adaptées aux flottants [20, 3]. La principale difficulté de cette démarche réside dans l'élaboration de fonctions de projections inverses conservatives de l'ensemble des solutions. En effet, si les projections directes, c.-à-d. le calcul du domaine de y à partir du domaine de x pour une contrainte du type $y = f(x)$, ne nécessitent qu'une légère adaptation des résultats classiques de l'arithmétique des intervalles, les projections inverses, c.-à-d. le calcul du domaine de x à partir de celui de y, ne répondent pas aux mêmes règles en raison des propriétés de l'arithmétique flottante. L'ensemble de ces résultats est détaillé dans [20] et étendu dans [3]. FPCS implémente aussi des consistances plus fortes comme les kB-consistances [18] plus à même de traiter les classiques problèmes d'occurrences multiples, mais aussi, de réduire de manière plus conséquente les bornes des domaines des variables.

Contrairement à la plupart des approches citées en section 2.2, les domaines des flottants manipulés par FPCS ne se réduisent pas aux valeurs numériques mais incluent aussi les infinis. De plus, le solveur FPCS traite l'ensemble des opérations arithmétiques de base ainsi que la plupart des fonctions mathématiques usuelles. Les conversions de type sont aussi correctement gérées.

Le comportement des programmes qui contiennent des calculs flottants peut varier avec le langage de programmation ou le compilateur, mais aussi avec le

[3] FLUCTUAT : http://www-list.cea.fr/labos/fr/LSL/fluctuat/index.html

[4] REALPAVER : http://pagesperso.lina.univ-nantes.fr/info/perso/permanents/granvil/realpaver/

système d'exploitation ou l'architecture d'exécution. FPCS cible des programmes C, compilés avec GCC sans option d'optimisation et destinés à une architecture x86 gérée par un système d'exploitation Linux 32 bits.

3.4 Mise en œuvre

Nous pouvons maintenant détailler notre approche. Les étapes du processus mis en œuvre dans l'approche que nous proposons sont les suivantes :

1. Pour un programme C donné, nous calculons avec FLUCTUAT une première approximation des domaines des variables sur les réels et sur les flottants.
2. Nous analysons le programme C et construisons deux systèmes de contraintes pour chaque chemin d'exécution atteignable (voir détails ci-dessous) : l'un avec des variables réelles et l'autre avec des variables flottantes. Nous affectons les domaines estimés par FLUCTUAT aux variables du programme.
3. Pour chaque système de contraintes, nous filtrons les domaines avec une consistance partielle.
   Nous utilisons FPCS et une $3B(w)$-consistance sur les systèmes avec les variables flottantes.
   Nous utilisons REALPAVER et sa consistance $BC5$ sur les systèmes avec les variables réelles. La $BC5$-consistance combine la méthode de Newton sur les intervalles et les *box* et *hull*-consistances.
4. Pour chaque variable de sortie, nous calculons l'union de tous les domaines obtenus à l'étape 3.

Lors de l'étape 2, nous explorons chaque chemin d'exécution d'un programme, c.-à-d. chaque chemin du graphe de flot de contrôle du programme, par une analyse en avant (du début à la fin du programme). Les instructions sont converties en forme SSA (*Static Single Assignment*), dans laquelle chaque variable n'est affectée qu'une seule fois dans chaque chemin d'exécution [8]. La longueur des chemins est bornée car les appels récursifs de fonction ne sont pas autorisés et les boucles sont dépliées un nombre fini de fois précisé par l'utilisateur. Les états possibles du programme en tout point d'un chemin d'exécution sont représentés par un système de contraintes composé d'un ensemble fini de variables et de contraintes sur ces variables. Avec FPCS, les variables ont des domaines qui correspondent aux implémentations en machine des types du langage C (`int`, `float` et `double`) ; avec REALPAVER, les domaines sont des intervalles sur les réels. Des règles définissent comment chaque instruction d'un programme modifie les états possibles du programme en ajoutant de nouvelles contraintes et variables.

Les chemins d'exécution sont explorés à la volée et interrompus dès que l'inconsistance du système de contraintes associé est détectée : simple filtrage par $3B$-consistance avec FPCS et par *hull*-consistance $HC4$ avec REALPAVER. Ceci permet de limiter l'explosion combinatoire du nombre des chemins en n'explorant que ceux qui sont exécutables. Cette technique de représentation des programmes par des systèmes de contraintes a été introduite pour la vérification bornée des programmes avec CPBPV [5]. Son implémentation pour l'approche proposée dans cet article s'appuie sur les bibliothèques développées pour CPBPV.

Le langage de modélisation de REALPAVER ne dispose pas des opérateurs d'inégalité stricte. Ceux-ci peuvent néanmoins apparaître dans les expressions conditionnelles des programmes. Par conséquent, dans les systèmes de contraintes générés pour REALPAVER, les inégalités strictes sont remplacées par leur équivalent non strict et les contraintes contenant l'opérateur « différent de » sont ignorées. Ceci peut conduire à des sur-approximations, mais aucune solution n'est perdue.

4 Expérimentations et discussion

Dans cette section, nous présentons les résultats de notre approche sur des programmes caractéristiques qui montrent quand notre approche améliore l'approximation calculée par FLUCTUAT. Les résultats ont été obtenus avec une machine Linux sur architecture Intel Core 2 Duo à 2,8 GHz avec 4 Go de mémoire. Nous avons utilisé FLUCTUAT version 3.8.22_opt et REALPAVER version 0.4. Dans les tables qui suivent, les domaines réels et flottants sont dénotés par les symboles $\mathbb{R}$ et $\mathbb{F}$ respectivement. Les nombres sont arrondis à trois chiffres après la virgule par souci de lisibilité.

4.1 Instructions conditionnelles

Nous avons présenté le problème de l'intersection de domaines pour les analyses abstraites en section 2.1. Ici, nous illustrons le comportement de notre approche dans cette situation sur le programme du calcul des racines des équations du second degré. Ce programme est donné en figure 2 et a été extrait de la bibliothèque scientifique GNU (GSL [23]). Les racines sont calculées dans les variables `x0` et `x1`. Le programme fait appel à deux fonctions de la bibliothèque C : `fabs` et `sqrt`, pour la valeur absolue et la racine carrée d'un nombre, respectivement. La fonction `sqrt` est d'ailleurs l'une des seules fonctions à être définies par le standard IEEE 754. Ces fonctions sont directement gérées par FPCS et peuvent donc apparaître telles quelles dans des contraintes. REALPAVER dispose d'une fonction

```
int quadratic(double a, double b, double c) {
  double r, sgnb, temp, r1, r2
  double disc = b * b - 4 * a * c;
  if (a == 0) {
    if (b == 0)
      return 0;
    else {
      x0 = -c / b;
      return 1;
    }
  }
  if (disc > 0) {
    if (b == 0) {
      r = fabs (0.5 * sqrt (disc) / a);
      x0 = -r;
      x1 = r;
    } else {
      sgnb = (b > 0 ? 1 : -1);
      temp = -0.5 * (b + sgnb * sqrt (disc));
      r1 = temp / a;
      r2 = c / temp;
      if (r1 < r2) {
        x0 = r1;
        x1 = r2;
      } else {
        x0 = r2;
        x1 = r1;
      }
    }
    return 2;
  } else if (disc == 0) {
    x0 = -0.5 * b / a;
    x1 = -0.5 * b / a;
    return 2;
  } else
    return 0;
}
```

FIG. 2 – Équation du second degré

`sqrt` prédéfinie, en revanche `fabs(x)` a été remplacé par `max(x,-x)`.

Nous donnons dans la table 2 les temps d'analyse et les approximations des domaines des variables `x0` et `x1` obtenus avec deux configurations des domaines des variables d'entrée. Les deux premières lignes de la table présentent les résultats de FLUCTUAT et de REALPAVER sur les réels. Les deux lignes suivantes présentent les résultats de FLUCTUAT et de FPCS sur les flottants. Dans la première configuration, où $a \in [-1 .. 1]$, $b \in [0{,}5 .. 1]$ et $c \in [0 .. 2]$, la sur-approximation de FLUCTUAT est tellement importante qu'elle n'apporte aucune information sur le domaine des racines, alors que notre approche est capable de réduire significativement ces domaines à la fois sur $\mathbb{R}$ et sur $\mathbb{F}$. Néanmoins, le problème d'intersection des domaines a parfois moins de conséquences sur les bornes de tous les domaines. Ceci est illustré par le domaine sur $\mathbb{F}$ de `x0` dans la seconde configuration où $a, b, c \in [1 .. 1 \cdot 10^6]$. Bien que le domaine calculé par FLUCTUAT soit une sur-approximation, notre approche ne parvient pas à le réduire. En revanche, dans cette même configuration, notre approche réalise de nouveau une très bonne réduction pour le domaine de `x1`.

Pour augmenter la précision de l'analyse, FLUCTUAT offre la possibilité de diviser les domaines d'au plus deux des variables d'entrée en un nombre paramétrable de sous-domaines. Les analyses sont alors effectuées sur chaque combinaison de sous-domaines et les résultats sont rassemblés. Sans connaissance a priori des domaines à subdiviser, toutes les combinaisons à un, puis à deux domaines doivent être essayées. Le nombre de subdivisions de chaque domaine est lui aussi difficile à déterminer ; il doit être choisi le plus grand possible tout en maintenant un temps d'analyse acceptable, et ceci sans garantie d'améliorer la précision. Pour la table 3, nous avons fixé les subdivisions à 50 lorsqu'un seul domaine est divisé et à 30 pour chaque domaine sinon. Nous ne présentons que les résultats sur $\mathbb{F}$ dans la table. Sur $\mathbb{R}$, dans la première configuration, les subdivisions n'apportent aucune amélioration ; dans la seconde configuration, les résultats sont identiques à ceux sur $\mathbb{F}$. Le coût en temps de ces subdivisions peut être important au regard du gain en précision obtenu :

- Dans la première configuration, les subdivisions du domaine de a entraînent une réduction intéressante du domaine de `x0` (identique à celle obtenue avec notre approche). Aucune combinaison de subdivisions ne réduit le domaine de `x1`.
- Dans la seconde configuration, la meilleure réduction du domaine de `x1` est obtenue en subdivisant à la fois les domaines de a et de b. Le gain reste toutefois assez faible par rapport à celui obtenu par notre approche. Aucune combinaison de subdivisions ne réduit le domaine de `x0`.

Lorsqu'il est nécessaire de subdiviser tous les domaines d'entrée, le coût devient prohibitif. Notre approche s'avère bien plus efficace : elle améliore souvent la précision de l'approximation et son coût est faible, que la précision soit améliorée ou non. En outre, la technique des subdivisions serait aussi applicable à notre approche.

4.2 Expressions non-linéaires

Le domaine abstrait utilisé par FLUCTUAT est basé sur des formes affines, qui ne permettent pas une représentation exacte des opérations non-linéaires : l'image d'un zonotope par une fonction non-linéaire n'est pas

		conf. #1 : $a \in [-1 .. 1]$ $b \in [0{,}5 .. 1]$ $c \in [0 .. 2]$			conf. #2 : $a, b, c \in [1 .. 1 \cdot 10^6]$		
		`x0`	`x1`	Temps	`x0`	`x1`	Temps
$\mathbb{R}$	FLUCTUAT	$[-\infty .. \infty]$	$[-\infty .. \infty]$	0,1 s	$[-2 \cdot 10^6 .. 0]$	$[-1 \cdot 10^6 .. 0]$	0,1 s
	REALPAVER	$[-\infty .. 0]$	$[-8{,}011 .. \infty]$	1,5 s	$[-1 \cdot 10^6 .. 0]$	$[-5{,}186 \cdot 10^5 .. 0]$	0,5 s
$\mathbb{F}$	FLUCTUAT	$[-\infty .. \infty]$	$[-\infty .. \infty]$	0,1 s	$[-2 \cdot 10^6 .. 0]$	$[-1 \cdot 10^6 .. 0]$	0,1 s
	FPCS	$[-\infty .. 0]$	$[-8{,}064 .. \infty]$	0,3 s	$[-2 \cdot 10^6 .. 0]$	$[-2\ 503{,}709 .. 0]$	0,3 s

TAB. 2 – Domaine des racines de la fonction `quadratic`

	conf. #1		conf. #2	
	`x0`	Temps	`x1`	Temps
a subdivisé	$[-\infty .. -0]$	> 1 s	$[-1 \cdot 10^6 .. 0]$	> 1 s
b subdivisé	$[-\infty .. \infty]$	> 1 s	$[-5 \cdot 10^5 .. 0]$	> 1 s
c subdivisé	$[-\infty .. \infty]$	> 1 s	$[-1 \cdot 10^6 .. 0]$	> 1 s
a et b subdivisés	$[-\infty .. -0]$	> 10 s	$[-1{,}834 \cdot 10^5 .. 0]$	> 10 s
a et c subdivisés	$[-\infty .. -0]$	> 10 s	$[-1 \cdot 10^6 .. 0]$	> 10 s
b et c subdivisés	$[-\infty .. \infty]$	> 10 s	$[-5 \cdot 10^5 .. 0]$	> 10 s

TAB. 3 – Domaines de FLUCTUAT sur $\mathbb{F}$ pour la fonction `quadratic` avec subdivisions

```
double sinus(double x) {
  return x - x*x*x/6 + x*x*x*x*x/120
         + x*x*x*x*x*x*x/5040;
}
```

FIG. 3 – sinus

un zonotope en général. Les opérations non-linéaires sont alors approximées en introduisant un terme d'erreur. Les solveurs de contraintes que nous utilisons gèrent mieux les expressions non-linéaires. Ainsi, sur l'exemple du développement limité à l'ordre 7 de la fonction sinus en figure 3, notre approche améliore significativement l'approximation de FLUCTUAT (cf. table 4, colonne `sinus`).

Néanmoins, les solveurs de contraintes que nous utilisons, eux aussi, approximent les expressions non-linéaires. Ceci est illustré par le polynôme de Rump (en figure 4) extrait de [22]. Ce polynôme assez singulier a été conçu pour faire apparaître un phénomène d'annulation catastrophique sur certaines architectures quelle que soit la précision des flottants. Ni REALPAVER ni FPCS ne parviennent à réduire les domaines calculés par FLUCTUAT (cf. table 4, colonne `rump`).

4.3 Instructions itératives

FLUCTUAT déplie les boucles un nombre borné de fois avant d'appliquer l'opérateur d'élargissement (*widening*) de l'interprétation abstraite (dix dépliages par défaut). L'opérateur d'élargissement permet d'atteindre un point fixe et de terminer rapidement. Cependant, cet opérateur peut conduire à de très grandes sur-approximations. Cette situation se produit lors de l'analyse de la valeur renvoyée par le programme `sqrt`, qui calcule une valeur approchée à $1 \cdot 10^{-2}$ de la racine carrée d'un nombre supérieur à 4. L'algorithme de `sqrt` est basé sur la méthode dite babylonienne (cf. figure 5). FLUCTUAT ne réalise aucune réduction du domaine de la valeur de retour de `sqrt` sur $\mathbb{F}$ pour la configuration où $x \in [5 .. 10]$ (cf. table 5).

```
double rump(double x, double y) {
  double f;
  f = 333.75*y*y*y*y*y*y*y;
  f = f + x*x*(11*x*x*y*y - y*y*y*y*y*y*y
               - 121*y*y*y*y - 2);
  f = f + 5.5*y*y*y*y*y*y*y*y*y;
  f = f + x / (2*y);
  return f;
}
```

FIG. 4 – Polynôme de Rump

Dans notre approche, nous n'essayons pas d'analyser le comportement des boucles : nous déplions simplement les boucles N fois, où N est fixé par l'utilisateur[5]. Nous n'essayons de réduire les domaines calculés par FLUCTUAT que si les conditions d'entrée des boucles sont fausses pour un nombre de dépliages k inférieur à N.

La table 5 présente les résultats que nous obtenons avec $N = 10$. Dans notre approche, les dé-

[5] Pour estimer une borne du nombre de dépliages nécessaires, nous pouvons aussi utiliser FLUCTUAT [14].

		`sinus` $x \in [-1 .. 1]$		`rump` $x \in [7 \cdot 10^4 .. 8 \cdot 10^4]$ $y \in [3 \cdot 10^4 .. 4 \cdot 10^4]$	
		Domaine	Temps	Domaine	Temps
$\mathbb{R}$	FLUCTUAT	$[-1{,}009 .. 1{,}009]$	0,1 s	$[-1{,}168 \cdot 10^{37} .. 1{,}992 \cdot 10^{37}]$	0,1 s
	REALPAVER	$[-0{,}842 .. 0{,}843]$	0,3 s	$[-1{,}144 \cdot 10^{36} .. 1{,}606 \cdot 10^{37}]$	1,2 s
$\mathbb{F}$	FLUCTUAT	$[-1{,}009 .. 1{,}009]$	0,1 s	$[-1{,}168 \cdot 10^{37} .. 1{,}992 \cdot 10^{37}]$	0,1 s
	FPCS	$[-0{,}853 .. 0{,}852]$	0,2 s	$[-1{,}168 \cdot 10^{37} .. 1{,}992 \cdot 10^{37}]$	0,2 s

TAB. 4 – Domaines de `sinus` et `rump`

		conf. #1 : $x \in [4{,}5 .. 5{,}5]$		conf. #2 : $x \in [5 .. 10]$	
		Domaine	Temps	Domaine	Temps
$\mathbb{R}$	FLUCTUAT	$[2{,}116 .. 2{,}354]$	0,1 s	$[2{,}098 .. 3{,}435]$	0,1 s
	REALPAVER	$[2{,}121 .. 2{,}346]$	0,3 s	$[2{,}232 .. 3{,}165]$	0,5 s
$\mathbb{F}$	FLUCTUAT	$[2{,}116 .. 2{,}354]$	0,1 s	$[-\infty .. \infty]$	0,1 s
	FPCS	$[2{,}120 .. 2{,}347]$	1 s	$[2{,}232 .. 3{,}168]$	1,6 s

TAB. 5 – Domaine de `sqrt`

```
double sqrt(double x) {
  double xn, xn1;
  xn = x/2;
  xn1 = 0.5*(xn + x/xn);
  while (xn-xn1 > 1e-2) {
    xn = xn1;
    xn1 = 0.5*(xn + x/xn);
  }
  return xn1;
}
```

FIG. 5 – Racine carrée

pliages peuvent rapidement devenir coûteux en temps, mais le gain en précision peut aussi être important : sur $\mathbb{F}$, dans la seconde configuration, nous calculons pour `sqrt` le domaine $[2{,}232 .. 3{,}168]$ au lieu du domaine $[-\infty .. \infty]$ obtenu par FLUCTUAT. REALPAVER est plus rapide que FPCS car nous employons avec lui une consistance plus faible pour détecter les chemins d'exécution inatteignables (*hull*-consistance pour REALPAVER *versus* $3B$-consistance pour FPCS). Notons que REALPAVER ne termine pas en un temps raisonnable si nous prenons $[-\infty .. \infty]$ comme domaine initial de la valeur de retour de `sqrt` au lieu du domaine calculé par FLUCTUAT.

4.4 Discussion

Les techniques d'interprétation abstraite calculent des approximations des domaines des variables sur une relaxation du problème initial. Dans le cas de FLUCTUAT, des ensembles de formes affines abstraient les expressions non-linéaires et les contraintes issues des programmes. Cette première approximation produit souvent des domaines suffisamment petits pour permettre un filtrage efficace avec des consistances partielles qui ne reposent pas sur la même relaxation.

Le filtrage par $3B$-consistance fonctionne bien avec FPCS. La $2B$-consistance n'est pas assez forte pour réduire les domaines calculés par FLUCTUAT, alors qu'une kB-consistance, plus forte, est trop coûteuse en temps. Nous avons aussi évalué plusieurs consistances implémentées dans REALPAVER. La table 6 présente les résultats les plus significatifs. $BC5$ est une combinaison de *hull*-consistance et de *box*-consistance avec la méthode de Newton sur les intervalles. $HC4$ est une *hull*-consistance sur les contraintes utilisateurs. Le délai maximum (T.O.) était fixé à 5 minutes. La $3B$-consistance est difficile à contrôler au moyen de la spécification de la largeur des sous-domaines à réfuter, c'est pourquoi nous avons introduit la version $3B$ temporisée : il s'agit de la $3B$-consistance interrompue après 0,5 s de filtrage. Il ressort que la $3B$-consistance est trop coûteuse en temps. Une consistance plus faible comme la $BC5$ représente un meilleur compromis entre temps d'exécution et réduction des domaines.

Bien que les mêmes approximations des domaines puissent parfois être obtenues sans partir de celles calculées par FLUCTUAT (c.-à-d. en prenant pour domaines initiaux $[-\infty .. \infty]$), nos expérimentations montrent que notre approche donne généralement de meilleurs résultats lorsque nous utilisons les approximations calculées par FLUCTUAT. Par exemple, pour la fonction `sqrt` de la figure 5, REALPAVER excède le délai maximum fixé pour filtrer les domaines si ceux-ci sont tous initialisés avec l'intervalle $[-\infty .. \infty]$; alors que le filtrage prend moins qu'une demi-seconde si les

	`quadratic` x1 conf. #1		`quadratic` x1 conf. #2		`sqrt` conf. #1	
	Domaine	Temps	Domaine	Temps	Domaine	Temps
$3B$	n.d.	T.O.	n.d.	T.O.	n.d.	T.O.
$3B$ temporisée (0,5 s)	$[-11{,}444 .. \infty]$	3 s	$[-749\ 999{,}721 .. 0]$	2,4 s	$[2{,}12 .. 2{,}346]$	1,3 s
weak$3B$	$[-8{,}004 .. \infty]$	9,3 s	$[-206\ 746{,}455 .. 0]$	5 s	$[2{,}121 .. 2{,}346]$	1 s
$BC5$	$[-8{,}011 .. \infty]$	1,5 s	$[-518\ 518{,}519 .. 0]$	0,5 s	$[2{,}121 .. 2{,}346]$	0,3 s
$HC4$	$[-8{,}223 .. \infty]$	0,8 s	$[-518\ 518{,}519 .. 0]$	0,5 s	$[2{,}121 .. 2{,}346]$	0,3 s

Tab. 6 – Comparaison des consistances de RealPaver

domaines sont initialisés avec les bornes calculées par Fluctuat. De même, pour la fonction `quadratic` de la figure 2, FPCS calcule le domaine $[-5{,}001 \cdot 10^{11} .. 0]$ en partant de $[-\infty .. \infty]$, alors que Fluctuat produit une meilleure approximation avec le domaine $[-2 \cdot 10^6 .. 0]$.

Bien sûr, les techniques de filtrage de la programmation par contraintes ne parviennent pas toujours à raffiner les approximations calculées par Fluctuat, en particulier lorsque la relaxation effectuée par Fluctuat se révèle appropriée pour calculer de bonnes approximations des domaines (e.g., avec les systèmes linéaires). De plus, même lorsque le filtrage serait en mesure de réduire les domaines, le temps de calcul peut être trop long, par exemple quand un grand nombre de dépliages des boucles est nécessaire ou quand des phénomènes de convergence lente se produisent dans FPCS [19].

5 Conclusion

Dans cet article, nous avons introduit une nouvelle approche pour raffiner les approximations des domaines des variables calculées par l'analyseur statique de programmes C Fluctuat. Cette approche repose sur des solveurs de contraintes, RealPaver et FPCS, qui sont corrects sur les nombres réels et les nombres flottants respectivement. Nous exploitons la capacité de réfutation des consistances partielles pour réduire les domaines calculés par Fluctuat. Nous avons montré que cette approche est rapide et efficace sur des programmes caractéristiques des difficultés de Fluctuat (constructions conditionnelles et non-linéaires).

Cependant, notre approche ne se substitue pas à Fluctuat. Les outils basés sur l'interprétation abstraite tels que Fluctuat calculent efficacement une approximation *globale* des domaines : en d'autres termes, l'interprétation abstraite d'un programme fusionne en chaque point du programme les domaines issus de l'analyse de tous les chemins du programme [6]. L'approximation globale des instructions conditionnelles et l'opération d'élargissement dans les instructions itératives facilitent le passage à l'échelle mais au prix de sur-approximations qui peuvent être très imprécises. D'autre part, les zonotopes constituent de meilleures approximations des systèmes de contraintes linéaires que les boîtes utilisées dans les solveurs de contraintes sur intervalles. Toutefois, les zonotopes sont moins bien adaptés aux systèmes de contraintes non-linéaires. Les techniques de filtrage employées dans des solveurs de systèmes de contraintes numériques offrent quant à elles un cadre flexible et extensible pour traiter les contraintes non-linéaires. L'exploration de chaque chemin d'exécution séparément est essentielle au calcul d'approximations précises. Cependant, afin de limiter l'explosion combinatoire, nous devons borner la longueur des chemins. Par conséquent, l'approche proposée dans cet article est complémentaire de celle de Fluctuat : notre approche donne de meilleurs résultats lorsque Fluctuat a réduit les domaines auparavant.

L'extension naturelle de ce travail est d'étudier comment Fluctuat peut être combiné plus étroitement avec des solveurs de contraintes. Des consistances plus fortes sont bien sûr envisageables pour améliorer encore la précision des approximations, e.g., des contraintes globales comme la *Quad* sur les expressions non-linéaires [17] ou la kB-consistance appliquée aux domaines des variables de sortie uniquement.

Remerciements

Les auteurs remercient Sylvie Putot et Éric Goubault du CEA-List pour leurs conseils et explications sur Fluctuat.

Références

[1] Ali Ayad and Claude Marché. Multi-prover verification of floating-point programs. In *IJCAR*, volume 6173 of *LNCS*, pages 127–141. Springer, 2010.

[2] Sylvie Boldo and Jean-Christophe Filliâtre. Formal verification of floating-point programs. In *18th IEEE Symposium on Computer Arithmetic*, pages 187–194. IEEE, 2007.

[3] Bernard Botella, Arnaud Gotlieb, and Claude Michel. Symbolic execution of floating-point computations. *Software Testing, Verification and Reliability*, 16(2) :97–121, 2006.

[4] Angelo Brillout, Daniel Kroening, and Thomas Wahl. Mixed abstractions for floating-point arithmetic. In *9th International Conference on Formal Methods in Computer-Aided Design*, pages 69–76. IEEE, 2009.

[5] Hélène Collavizza, Michel Rueher, and Pascal Van Hentenryck. A constraint-programming framework for bounded program verification. *Constraints Journal*, 15(2) :238–264, 2010.

[6] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *6th ACM Symposium on Principles of Programming Languages*, pages 269–282, 1979.

[7] Patrick Cousot, Radhia Cousot, Jérôme Feret, Antoine Miné, Laurent Mauborgne, David Monniaux, and Xavier Rival. Varieties of static analyzers : A comparison with ASTRÉE. In *1st Joint IEEE/IFIP Symposium on Theoretical Aspects of Software Engineering*, pages 3–20. IEEE, 2007.

[8] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4) :451–490, 1991.

[9] Florent de Dinechin, Christoph Quirin Lauter, and Guillaume Melquiond. Assisted verification of elementary functions using Gappa. In *ACM Symposium on Applied Computing*, pages 1318–1322. ACM, 2006.

[10] David Delmas, Eric Goubault, Sylvie Putot, Jean Souyris, Karim Tekkal, and Franck Védrine. Towards an industrial use of fluctuat on safety-critical avionics software. In *FMICS*, volume 5825 of *LNCS*, pages 53–69. Springer, 2009.

[11] Khalil Ghorbal, Eric Goubault, and Sylvie Putot. A logical product approach to zonotope intersection. In *CAV*, volume 6174 of *LNCS*, pages 212–226. Springer, 2010.

[12] David Goldberg. What every computer scientist should know about floating point arithmetic. *ACM Computing Surveys*, 23(1) :5–48, 1991.

[13] Eric Goubault and Sylvie Putot. Static analysis of numerical algorithms. In *SAS*, volume 4134 of *LNCS*, pages 18–34. Springer, 2006.

[14] Eric Goubault and Sylvie Putot. Static analysis of finite precision computations. In *VMCAI*, volume 6538 of *LNCS*, pages 232–247. Springer, 2011.

[15] Laurent Granvilliers and Frédéric Benhamou. Algorithm 852 : Realpaver : an interval solver using constraint satisfaction techniques. *ACM Transactions on Mathematical Software*, 32(1) :138–156, 2006.

[16] John Harrison. A machine-checked theory of floating-point arithmetic. In *TPHOLs*, volume 1690 of *LNCS*, pages 113–130. Springer-Verlag, 1999.

[17] Yahia Lebbah, Claude Michel, and Michel Rueher. A rigorous global filtering algorithm for quadratic constraints. *Constraints*, 10(1) :47–65, 2005.

[18] Olivier Lhomme. Consistency techniques for numeric CSPs. In *13th International Joint Conference on Artificial Intelligence*, pages 232–238, 1993.

[19] Bruno Marre and Claude Michel. Improving the floating point addition and subtraction constraints. In *CP*, volume 6308 of *LNCS*, pages 360–367. Springer, 2010.

[20] Claude Michel. Exact projection functions for floating-point number constraints. In *7th International Symposium on Artificial Intelligence and Mathematics*, 2002. `http://rutcor.rutgers.edu/~amai/aimath02/PAPERS/21.ps`.

[21] David Monniaux. The pitfalls of verifying floating-point computations. *ACM Transactions on Programming Languages and Systems*, 30(3) :12 :1–12 :41, 2008.

[22] Siegfried M. Rump. Verification methods : Rigorous results using floating-point arithmetic. *Acta Numerica*, 19 :287–449, 2010.

[23] The GSL Team. *GNU Scientific Library Reference Manual*, 1.14 edition, 2010. `http://www.gnu.org/software/gsl/`.

Résolution étendue et largeur de réfutation de formules SAT

Nicolas Prcovic
LSIS - Aix-Marseille Universités
nicolas.prcovic@lsis.org

Résumé

On appelle résolution de largeur k, une restriction de la résolution interdisant de produire des résolvantes de taille supérieure ou égale à $k+1$. Nous montrons que la résolution étendue (ie, incorporant la règle d'extension) de largeur 3 des instances 3-SAT est complète. De plus, nous montrons que le problème des pigeons peut se résoudre en temps polynômial grâce à la résolution étendue de largeur 3. Au final, nous obtenons donc un système de preuve dont la combinatoire est fortement réduite mais qui reste complet et plus efficace que la résolution sur certains problèmes.

Abstract

We call k-width Resolution, a restriction of Resolution forbiding the production of resolvents of size greater than or equal to $k+1$. We show that 3-width Extended Resolution (ie, Resolution incorporating the extension rule) on 3-SAT instances is complete. Furthermore, we show that the pigeonhole problem can be solved in polynomial time by 3-width Extended Resolution. We thus obtain a proof system whose combinatorics is greatly reduced but remains complete and more powerful than Resolution on some problems.

1 Introduction

Les systèmes de preuve basés sur la résolution sont utilisés pour permettre de trouver la réfutation d'une formule booléenne supposée insatisfiable. Cependant, certaines familles d'instances [7, 13, 4] nécessitent de produire un nombre exponentiel de résolvantes. Dans ces cas-là, beaucoup de résolvantes produites ont beaucoup de littéraux. La résolution étendue [12] ajoute une régle supplémentaire à la résolution, consistant à ajouter les trois clauses correspondant à $x \Leftrightarrow l_1 \vee l_2$ à la formule, où seule x est une nouvelle variable. Dans ce cadre, Cook [5] exhibe une réfutation de longueur polynômiale du problème des pigeons. Personne n'a jamais trouvé d'instances nécessitant la production d'un nombre exponentiel de résolvantes avec la résolution étendue. Mais bien que rendant un système de preuve par résolution plus puissant, la règle d'extension introduit une source supplémentaire d'explosion combinatoire. En effet, le nombre potentiel de résolvantes que l'on peut produire n'est plus une exponentielle du nombre de variables de la formule mais du nombre total de variables, qui inclut les variables ajoutées par les extensions. Or, le nombre d'extensions que l'on peut faire n'est pas nécessairement polynômial. C'est pourquoi les rares systèmes l'incluant (GUNSAT [2], GlucosEr [1], [8]) restreignent beaucoup l'application de cette règle afin que son avantage, le possible racourcissement de la longueur de la réfutation, ne soit pas contrebalancé par son inconvénient, le traitement des clauses supplémentaires. Dans cet article, nous proposons un changement de perspective sur la question : plutôt que de limiter l'application de la règle d'extension, nous limitons l'application de la règle de résolution à la production de clauses de taille 3 au maximum. Ce qui légitime cette approche est que l'ajout d'extensions est fait pour réduire la longueur de la preuve, donc il faut supposer que la taille des résolvantes sera petite.

Après avoir rappelé quelques notions sur le problème SAT (section 2), donné les rapports entre longueur et largeur de preuve (section 3) et introduit la résolution étendue (section 4), nous allons démontrer que, malgré cette restriction sur la largeur des preuves, ce système de preuve reste complet (section 5) et qu'il permet de donner une réfutation de longueur polynômiale du problème des pigeons (section 6).

2 Notions préliminaires

Une *formule booléenne*, sous sa forme dite *CNF*, est un ensemble de clauses. Une clause est un ensemble de littéraux. Un littéral est soit une variable booléenne, soit sa négation. On définit var(l) comme étant la variable du littéral l. Les variables peuvent être affectées à la valeur vrai ou faux. Une clause représente une disjonction de l'ensemble de ses littéraux. Une formule SAT représente une conjonction de ses clauses. Un *modèle* d'une formule ϕ est une affectation de variables qui est telle que ϕ est vraie. Une formule n'ayant pas de modèle est dite *insatisfiable*. La clause vide, notée $\square$, est toujours fausse et si une formule la contient alors elle est insatisfiable.

Les systèmes formels de preuve propositionnelle permettent de dériver d'autres formules à partir d'une formule ϕ, grâce à des règles d'inférence. En particulier, le système de Robinson [11], que nous appellerons `Res`, permet de dériver des clauses induites à partir des clauses d'une formule ϕ grâce à la règle de *résolution* :

$$C_1, C_2 \vdash C_1 \backslash \{l\} \cup C_2 \backslash \{\bar{l}\}$$

La clause inférée par résolution de C_1 avec C_2 sera appelé *résolvante* sur var(l) de C_1 et C_2. L'intérêt de `Res` est qu'il permet d'établir des réfutations de formules (ie, des preuves de leur insatisfiabilité) dans la mesure où une formule est insatisfiable si et seulement si il existe une suite de résolutions qui dérive la clause vide. `Res` fonctionne par saturation de l'application de sa règle : si une formule est satisfiable, `Res` le détecte quand plus aucune résolvante non redondante ne peut être dérivée.

Une dérivation est une suite d'applications de la règle de résolution permettant de dériver une clause. Elle constitue la preuve que cette clause est une conséquence logique de la formule. Elle peut se représenter grâce à un arbre binaire dont les feuilles sont des clauses de ϕ, les nœuds internes sont des résolvantes et la racine est la clause dérivée. Une dérivation de la clause vide s'appelle une *réfutation*.

3 Longueur et largeur de preuve

On appelle *longueur d'une preuve*, le nombre de résolvantes qu'elle contient. On appelle *taille* d'une clause le nombre de littéraux qu'elle contient. On appelle *largeur d'une preuve* la taille de la plus grande clause apparaissant dans cette preuve.

Les liens entre longueur et largeur de preuve sont maintenant assez bien établis. Il est clair qu'une preuve étroite est forcément courte. Plus précisément, si la largeur de la preuve est k alors la preuve ne peut contenir que de l'ordre de $\Theta(n^k)$ résolvantes différentes, où n est le nombre de variables de la formule. Si k est une constante alors la longueur de la preuve est polynômialement bornée. Dans [3], les auteurs montrent complémentairement que les preuves courtes sont forcément étroites. A partir de cette relation entre largeur et longueur de preuve, il est possible de définir des algorithmes de recherche de réfutations qui restreignent la largeur des résolvantes produites. Par exemple, le pré-traitement de Satz[10] (avant une résolution complète à la DPLL) consiste à dériver toutes les résolvantes possibles de taille 3 au maximum jusqu'à saturation (ou dérivation éventuelle de la clause vide). Plus généralement, une manière de garantir une complexité polynômiale consiste à restreindre la résolution à la production de résolvantes de taille inférieure à une borne donnée. Evidemment, cela se fait au détriment de la complétude dans la mesure où il est parfois nécessaire de dériver des grandes clauses avant d'obtenir la clause vide. On peut aussi définir une méthode complète qui restreint la résolution à la dérivation de résolvantes de taille k maximum mais qui incrémente la valeur de k à chaque fois qu'il y a saturation [3]. Dans cet article, nous explorons une autre voie. Au lieu d'augmenter la valeur de k, que nous fixons une fois pour toute à 3, nous nous plaçons dans la cadre de la résolution étendue, qui va nous permettre de rajouter des clauses permettant de continuer à ne dériver que d'autres résolvantes de taille 3.

4 La résolution étendue

La résolution étendue [12] consiste à ajouter à `Res` la *règle d'extension* en plus de la règle de résolution. Nous appellerons `ER` ce nouveau système de preuve. En toute généralité, la règle d'extension permet d'ajouter (à l'ensemble des clauses) les clauses correspondant à la formule $x \Leftrightarrow F$, où x est une nouvelle variable et F est n'importe quelle formule portant sur des variables existant déjà. A partir de maintenant, nous distinguerons les nouvelles variables introduites par des extensions, dites *variables d'extension*, des variables de la formule initiale, dites *variables d'entrée*.

On peut toujours se restreindre à ce que F soit uniquement du type $l_1 \vee l_2$ où l_1 et l_2 sont des littéraux portant sur des variables déjà présentes. En effet, il est facile de décomposer récursivement n'importe quelle formule F en une composition de disjonctions de sous-formules, éventuellement négatives. En associant chaque sous-formule à une nouvelle variable, on obtient un ensemble d'extensions de type $x_i \Leftrightarrow (l_1 \vee l_2)$ au lieu d'une seule du type $x \Leftrightarrow F$. Ce type d'extension portant sur la disjonction de deux littéraux préexistant sera appelée *extension binaire disjonctive*. Une extension $x \Leftrightarrow (l_1 \vee l_2)$ consiste à ajouter les clauses

$\overline{x} \vee l_1 \vee l_2$, $x \vee \overline{l_1}$ et $x \vee \overline{l_2}$. Il faut noter qu'il est aussi parfaitement possible de se restreindre à des extensions binaires conjonctives (i.e., de type $x \Leftrightarrow (l_1 \wedge l_2)$, qui consiste à ajouter les clauses $x \vee \overline{l_1} \vee \overline{l_2}$, $\overline{x} \vee l_1$ et $\overline{x} \vee l_2$).

L'intérêt de rajouter la règle d'extension est qu'elle rend le système plus puissant dans la mesure où certains problèmes dont la preuve est de longueur nécessairement exponentielle dans `Res` ont une preuve polynômiale dans `ER`. C'est le cas du fameux problème des pigeons [7, 5]. En dehors de ce problème particulier, on peut identifier des cas plus généraux où une extension permet un racourcissement de preuve. Considèrons une preuve contenant des séquences de résolutions qui font que $C_1 \vee C$ est utilisée pour dériver $C_1 \vee C'$ et que $C_2 \vee C$ est utilisée pour dériver $C_2 \vee C'$ grâce à la même suite de clauses qui permet de passer de C à C' (C_1, C_2, C et C' sont des clauses). On peut vérifier que grâce aux clauses émanant de $x \Leftrightarrow C_1 \wedge C_2$, on dérive d'abord $x \vee C$ à partir de $C_1 \vee C$ et $C_2 \vee C$, puis on effectue *une seule fois* la séquence de résolutions sur C permettant de dériver $x \vee C'$, à partir de quoi on peut à nouveau dériver $C_1 \vee C'$ et $C_2 \vee C$ (encore grâce aux clauses émanant de la même extension). Cette idée est mise en œuvre explicitement dans [1], que les auteurs appellent *compression de preuve* (et implicitement dans [8]) lors de la phase d'apprentissage d'une clause dans un solveur CDCL. Dans cet article, nous nous plaçons dans le cadre de la résolution étendue à l'aide d'extensions binaires disjonctives, qui permet un autre type de transformation de preuve, comme nous le verrons dans la section suivante.

Une question est de savoir quelles extensions il est intéressant d'effectuer afin de réduire la longueur d'une preuve. Mais une question encore plus cruciale est de savoir quelles extensions il est toujours inutile d'effectuer. Nous abordons ce point maintenant.

Définition 1 *Développement d'une extension*
On appelle développement d'une extension $x \Leftrightarrow F$, *l'expression d'une extension dans laquelle on remplace récursivement chaque occurence de variable d'extension de F par la disjonction binaire de l'extension qui l'a introduite, jusqu'à ce que F ne contiennent plus que des littéraux portant sur des variables d'entrée.*
Le développement CNF d'une extension $x \Leftrightarrow F$ *consiste à la mise sous forme CNF du développement de F.*

Exemple : étant données les extensions $x \Leftrightarrow (a \vee b)$ et $y \Leftrightarrow (\overline{x} \vee c)$, le développement de $z \Leftrightarrow (\overline{y} \vee d)$ est $z \Leftrightarrow (\overline{\overline{a \vee b} \vee c} \vee d)$ et son développement CNF est $z \Leftrightarrow ((a \vee b \vee d) \wedge (\overline{c} \vee d))$.

Définition 2 *Niveau d'une variable, d'une extension*
Etant donnée une formule ϕ, on définit récursivement le niveau *N(x) d'une variable x ainsi :*
- *Si x est une variable d'entrée, $N(x) = 0$.*
- *Soit x la variable introduite par l'extension $x \Leftrightarrow (l_1 \vee l_2)$. On a $N(x) = max(N(var(l_1)), N(var(l_2))) + 1$.*

Le niveau d'une extension *est égal au niveau de la variable qu'elle introduit.*

Proposition 1 *n étant le nombre de variables d'entrée, si le niveau d'une extension est supérieur à $n \log n$ alors il existe une extension dont le développement CNF est identique mais dont le niveau est inférieur.*

Preuve 1 *On peut construire n'importe quel développement CNF contenant n variables comme composition récursive de disjonctions binaires. Pour toute formule CNF à n variables, chacune de ses clauses contient au plus n littéraux et la formule contient au plus 2^n clauses. Par dichotomie, on peut exprimer n'importe quel clause de taille n grâce à une composition récursive de clauses binaires. La hauteur de cette décomposition est $\log n$. De même, par dichotomie, on peut représenter un ensemble de 2^n clauses en une composition récursive de conjonctions binaires de clauses. La négation de cette composition récursive est une composition récursive de disjonctions de négations de clauses. La hauteur de cette composition récursive est n. Donc en tout, la hauteur maximale de décomposition disjonctive récursive d'une formule CNF est $n \log n$. Comme on peut associer une variable d'extension à chaque disjonction binaire représentant une sous-formule, une extension dont le développement CNF porte sur n variables ne nécessite qu'un niveau de $n \log n$ au maximum.*

Puisqu'on peut se limiter à n'ajouter que des extensions de niveau inférieur à $n \log n$, le nombre d'extensions possibles non redondantes (ie, dont le développement CNF diffère de celui des autres extensions) est borné.

5 Résolution étendue et largeur de réfutation restreinte

Bien que `ER` soit théoriquement plus puissant que `Res`, i.e. permet de trouver des preuves plus courtes, en pratique elle pose des difficultés qui l'ont empêché jusqu'à récemment d'être performante. Le choix des extensions à effectuer est difficile à faire. Et quand bien même les bons choix seraient faits, les résultats ne seraient pas automatiquement meilleurs. Par exemple, quand nous avons ajouté les $\Theta(n^3)$ clauses issues de la règle d'extension proposés par Cook aux $\Theta(n^2)$ clauses

du problème des n pigeons, nous avons expérimenté qu'aucun type de solveurs (qu'il soit "moderne", que ce soit DP60 [6] ou de la SL-résolution [9]) ne résolvait plus rapidement (bien au contraire) le problème alors que son temps de résolution devenait théoriquement polynômial. A moins d'avoir un bon moyen de choisir les résolvantes à produire, il y a le risque de produire à peu près les mêmes résolvantes qu'auparavant et beaucoup d'autres encore. Notre idée est donc simple : puisque nous rajoutons des extensions pour que la longueur de la preuve soit courte, nous devons supposer que la largeur de la preuve sera petite.

Le principe est de faire de la résolution étendue sans jamais produire de clause de taille supérieure ou égale à 4. Nous allons d'abord présenter sa réalisation à travers un exemple très simple. Soit la formule $\phi = \{a \vee b \vee c, \bar{a} \vee d \vee e, \bar{b} \vee d \vee e\}$. Pour obtenir la résolvante $c \vee d \vee e$, on est obligé de générer une résolvante quaternaire (ie, de taille 4) (cf le 1er arbre de la figure 1). Or, en ajoutant une extension $x \Leftrightarrow (d \vee e)$ ou $x \Leftrightarrow (c \vee d)$, on peut dériver $c \vee d \vee e$ uniquement via des résolvantes de taille 3 (cf les deux derniers arbres de la figure 1).

Plus fondamentalement, l'idée est que lorsqu'une résolution entre deux clauses ternaires $a \vee b \vee c$ et $\bar{c} \vee d \vee e$ génère une clause quaternaire $a \vee b \vee d \vee e$, on va faire en sorte de se ramener à une dérivation de clause ternaire. Soit $l_1, l_2 \in \{a, b, d, e\}$ et $\{l_3, l_4\} = \{a, b, d, e\} \setminus \{l_1, l_2\}$. Grâce aux clauses $x \vee \bar{l_1}$ et $x \vee \bar{l_2}$ de l'extension $x \Leftrightarrow (l_1 \vee l_2)$, nous dérivons la clause devenue ternaire $x \vee l_3 \vee l_4$ car $l_1 \vee l_2$ a été remplacé par x. Ensuite, nous attendons qu'on ait dérivé une clause binaire $x \vee l$ à partir de $x \vee l_3 \vee l_4$ pour pouvoir remplacer x par $l_1 \vee l_2$ (une fois qu'il ne peut plus produire de clause quaternaire) : la résolution entre $x \vee l$ et $\bar{x} \vee l_1 \vee l_2$ produit alors $l \vee l_1 \vee l_2$.

La figure 2 montre les deux types de dérivations possibles : celle où l_1 et l_2 sont dans la même clause (en haut) et celle où l_1 et l_2 sont dans deux clauses différentes (en bas).

Nous verrons qu'il est toujours possible d'éviter de produire des clauses quaternaire de cette façon tout en restant complet. Mais ceci n'est d'abord possible que si les formules CNF que nous cherchons à réfuter sont des instances 3-SAT (i.e., ne contiennent que des clauses de taille 3). Dans le cas contraire, il est toujours possible de ramener une formule CNF à une instance 3-SAT équivalente du point de vue de la satisfiabilité. La technique standard consiste à remplacer une clause $C = a_1 \vee a_2 \vee ... \vee a_k$ par $k-2$ clauses ternaires $a_1 \vee a_2 \vee \overline{x_2}$, $x_2 \vee a_3 \vee \overline{x_3}$, ..., $x_{k-3} \vee a_{k-2} \vee \overline{x_{k-2}}$ et $x_{k-2} \vee a_{k-1} \vee a_k$, où les $k-3$ variables x_i sont nouvelles. Cela se fait simplement en ajoutant les extensions $x_2 \Leftrightarrow (a_1 \vee a_2)$ et $\forall i, 3 \leq i < k-1$, $x_i \Leftrightarrow (a_{i-1} \vee x_{i-1})$. En effet, il suffit

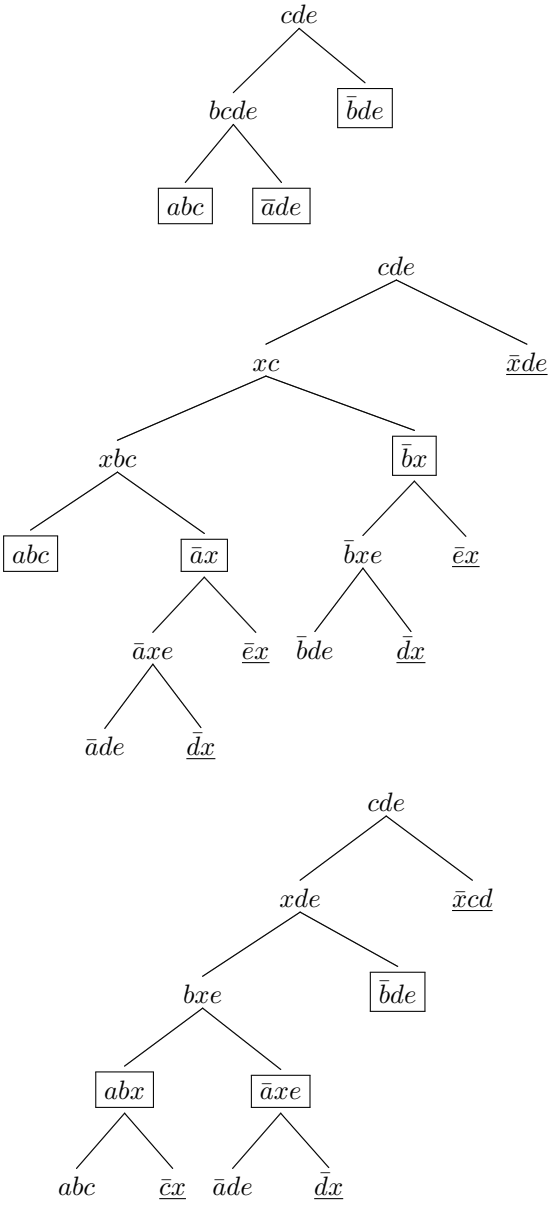


FIG. 1

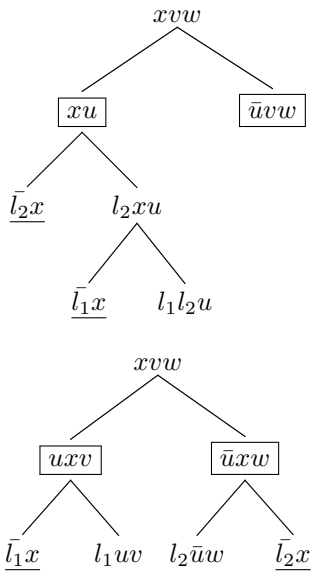


FIG. 2

de remarquer que les clauses ternaires ainsi ajoutées sont celles du type $x_i \vee a_{i+1} \vee \overline{x_{i+1}}$ qui remplacent C, tandis qu'en appliquant linéairement la suite de $2k-4$ résolutions entre les clauses binaires ajoutées et C, on dérive la dernière clause de remplacement $x_{k-2} \vee a_{k-1} \vee a_k$.

Définition 3 *Généalogie d'un littéral pour une résolvante*
Dans une preuve par résolution, la généalogie G d'un littéral x pour une résolvante C se définit inductivement ainsi :
- *$C \in G$.*
- *Si $C_1 \in G$ et que C_1 est la résolvante de C_2 et d'une autre clause et que x est un littéral de C_2 alors $C_2 \in G$.*

Définition 4 *Source de généalogie*
Dans une preuve par résolution, une source de généalogie G est une clause de G qui est aussi une clause d'entrée.

L'ensemble des sources d'une généalogie G d'une preuve faite à partir d'une formule ϕ est donc $G \cap \phi$.

Définition 5 *Amincissement de preuve arborescente*
L'amincissement d'une preuve (par résolution) arborescente, dont la racine et les feuilles sont de taille inférieure ou égale à 3 et dont au moins une des résolvantes est de taille 4, consiste en sa modification de la façon suivante. Considérons la racine de la preuve et remontons éventuellement l'un des chemins qui y mène jusqu'à trouver la première clause de taille 4, disons $a \vee b \vee c \vee d$. Sans perte de généralité, considérons que cette clause permet de produire la résolvante $b \vee c \vee d$ (avec une clause contenant $\overline{a}$ et éventuellement un certain nombre de littéraux parmi b, c et d). Considérons l'extension $x \Leftrightarrow (b \vee c)$ qui correspond aux clauses $x \vee \overline{b}$, $x \vee \overline{c}$ et $\overline{x} \vee b \vee c$. La modification de la preuve consiste à :
- *insérer une résolution entre $x \vee \overline{b}$ et chacune des sources de la généalogie de b pour la clause $b \vee c \vee d$. Ceci a pour effet de remplacer toutes les occurences de b par x dans toutes les clauses de la généalogie (à part les sources). En particulier, $b \vee c \vee d$ devient $x \vee c \vee d$.*
- *insérer une résolution entre $x \vee \overline{c}$ et chacune des sources de la généalogie de c pour la clause $x \vee c \vee d$. Ceci a pour effet de remplacer toutes les occurences de c par x dans toutes les clauses de la généalogie (à part les sources). En particulier, $x \vee c \vee d$ devient $x \vee d$ (les deux occurences de x fusionnent.*
- *insérer une résolution entre la résolvante $x \vee d$ et $\overline{x} \vee b \vee c$ pour obtenir à nouveau $b \vee c \vee d$.*

Un exemple d'amincissements se trouve en figure 3. Dans toutes les figures, nous soulignons les clauses qui sont ajoutées par les extensions.

Proposition 2 *Un amincissement de preuve arborescente contient au moins une clause de taille supérieure ou égale à 4 de moins qu'avant l'amincissement.*

Preuve 2 *Nous considérons l'amincissement tel qu'il est présenté dans la définition précédente. Il y a toujours une clause quaternaire $a \vee b \vee c \vee d$ s'il y a des clauses de taille supérieur à 4. En effet, une résolution ne peut produire une résolvante dont la taille est inférieure de plus d'une unité à la plus petite des deux clauses qu'on résout. Donc en partant d'une clause de taille supérieure à 4, on est obligé de passer par une clause de taille 4 pour aboutir à une clause de taille inférieure à 4.*

L'insertion des clauses $x \vee \overline{b}$ et $x \vee \overline{c}$ dans la preuve permet de renommer toutes les occurences de b et de c par x dans toutes les clauses (à part les sources) de la généalogie de a et b pour la clause $a \vee b \vee c \vee d$. Plus précisément, dans le cas où une clause contenait $b \vee c$, ce dernier est remplacé par x, ce qui réduit sa taille d'une unité. C'est le cas au moins de $a \vee b \vee c \vee d$ (de taille 4), qui devient $a \vee x \vee d$ (de taille 3). Par ailleurs, aucune autre clause de taille supérieure ou égale à 4 n'a pu apparaître car seules une clause ternaire et des clauses binaires ont été introduites, que l'effet de la clause ternaire a été de produire une clause ternaire tandis que l'effet des clauses binaires a été de renommer des littéraux.

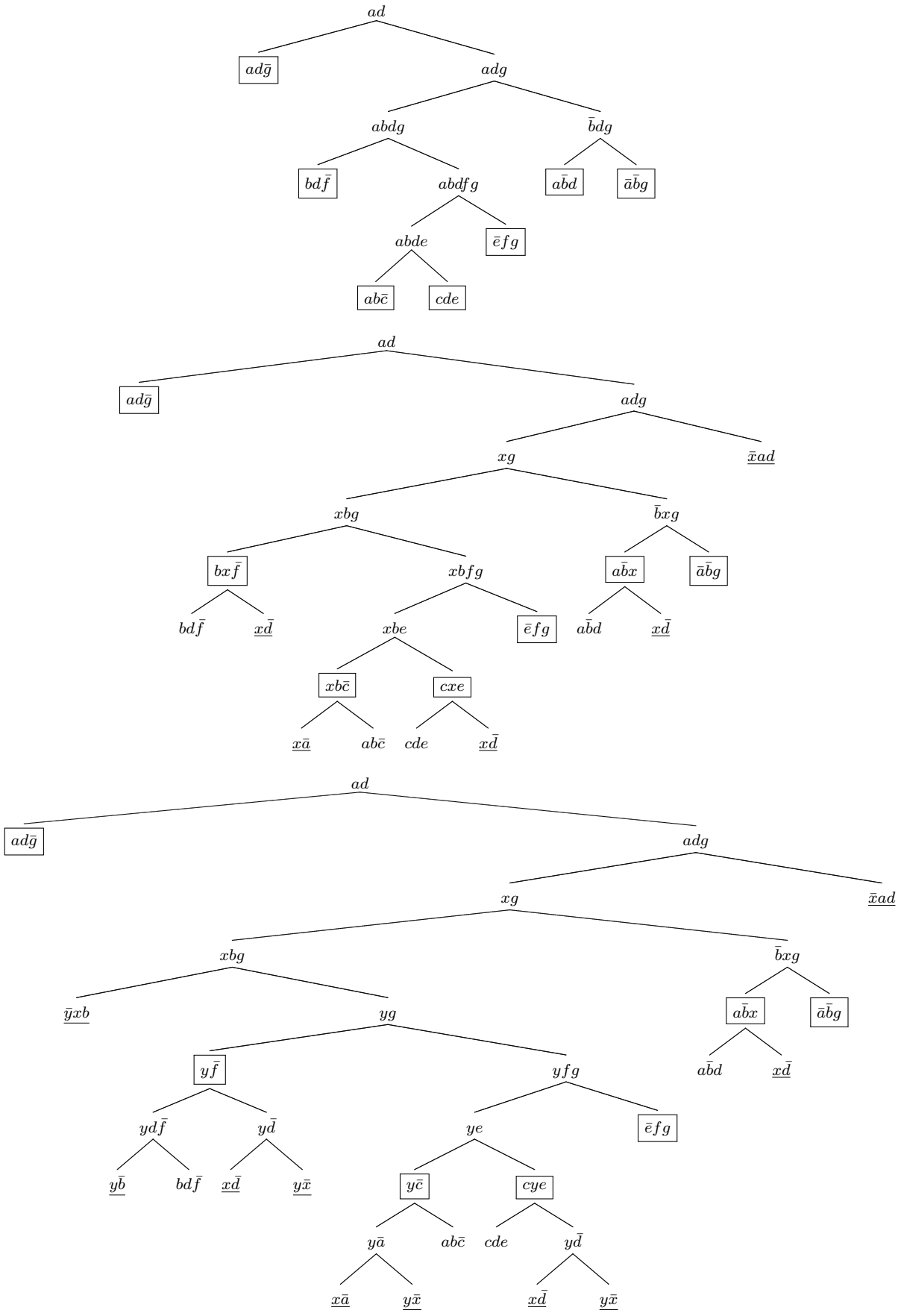


FIG. 3: Deux amincissements successifs d'une dérivation arborescente menant à une dérivation de largeur 3. Les deux extensions sont $x \Leftrightarrow (a \vee d)$ et $y \Leftrightarrow (x \vee b)$.

Proposition 3 *La résolution étendue de largeur 3 de formules 3-SAT est complète.*

Preuve 3 *Nous partons du fait qu'il existe une réfutation arborescente par résolution pour toute instance 3-SAT insatisfiable. D'après la proposition 2, un amincissement permet de décroître d'au moins une unité le nombre de résolvantes de taille supérieure ou égale à 4 d'une preuve dont la clause dérivée est de taille stritement inférieure à 4. En réitérant cette procédure d'amincissement, on élimine petit à petit toutes les résolvantes de taille 4 ou plus jusqu'à obtenir une réfutation de largeur 3.*

Nous appellerons ER3 le système de preuve ER restreignant la dérivation aux résolvantes de taille 3 maximum.

Nous nous intéressons maintenant à la nouvelle longueur des preuves (i.e., le nombre de nœuds des arbres) obtenues après une suite d'amincissements.

Lorsqu'on effectue un premier amincissement grâce à une extension $x \Leftrightarrow (l_1 \vee l_2)$, on insère une résolution avec $\overline{x} \vee l_1 \vee l_2$ et un certain nombre de résolutions entre une clause binaire, $x \vee \overline{l_1}$ ou $x \vee \overline{l_2}$, et une feuille de l'arbre de preuve. Le second type de résolution remplace une feuille contenant une clause C de l'arbre par un sous-arbre composé d'une racine et de deux feuilles : C et la clause binaire. La racine de ce sous-arbre est la clause C dont un littéral (l_1 ou l_2) a été remplacé par x. Lors des amincissements suivants, ce sous-arbre pourra aussi être modifié par des insertions de résolutions avec des clauses binaires. Ce qui était initialement une feuille (avant le premier amincissement) est devenu un sous-arbre dont la racine contient la clause de cette feuille dont les littéraux ont été renommés par les insertions de résolutions.

Définition 6 *Sous-arbre de renommage*
Un sous-arbre de renommage *est un sous-arbre de preuve représentant la façon dont une feuille d'un arbre de preuve a eu ses littéraux renommés par une suite d'amincissements.*

Il faut remarquer qu'un littéral peut être renommé en un littéral déjà présent dans la même clause. Dans ce cas, les littéraux fusionnent et la clause diminue sa taille d'une unité. En conséquence, la clause étant à la racine d'un sous-arbre de renommage peut être plus courte que la clause initiale dont elle représente le résultat des renommages successifs. Dans toutes les figures, nous avons encadré les racines des arbres de renommages.

Proposition 4 *Lorsqu'une suite d'amincissements d'une preuve arborescente réduit sa largeur à 3, la longueur de cette preuve est inférieur à $(3L + 1)L$, où L est la longueur de la preuve avant amincissements.*

Preuve 4 *Chaque amincissement dimininuant d'au moins une unité le nombre de résolvantes au moins quaternaires, le nombre d'amincissements à effectuer est inférieur à L, la longueur de la preuve (le nombre de nœuds de l'arbre) initiale. Le nombre global d'insertions de résolutions avec une clause ternaire de type $\overline{x} \vee l_1 \vee l_2$ est donc inférieur à L (car il y en a une par amincissement). Evaluons maintenant le nombre global d'insertions de résolutions avec des clauses binaires de type $x \vee \overline{l_1}$ (ou $x \vee \overline{l_2}$). Il est égal à la somme du nombre de nœuds des sous-arbres de renommage. Or, à chaque amincissement, chaque sous-arbre de renommage pourra éventuellement intégrer au maximum trois résolutions. Voici pourquoi.*
Au départ, un sous-arbre de renommage T ne contient qu'une feuille correspondant à une clause ternaire. Après un amincissement impliquant une extension $x \Leftrightarrow (l_1 \vee l_2)$, T n'est modifié que si sa racine contient l_1 (ou l_2). En effet, une résolution n'est inséré à partir d'une feuille de T que si c'est une source de généalogie de l_1 (ou l_2) et donc tous les nœuds sur le chemin entre cette feuille et le nœud contenant la clause de taille 4 qu'on veut amincir doivent contenir l_1 (ou l_2), ce qui doit donc être le cas de la racine de T. Si la racine de T contient à la fois l_1 et l_2 alors ces deux littéraux seront remplacés par une seule occurence de x et la taille de la clause à la racine sera décrémentée. Quel que soit le nombre d'amincissements, cela ne peut arriver que deux fois par sous-arbre de renommage : une première fois la clause ternaire à la racine devient binaire et une deuxième fois elle devient unaire. Examinons maintenant les cas où un seul des littéraux parmi l_1 et l_2 est présent dans la racine de T, disons l_1. Trois cas sont possibles :

- *La clause de la racine de T est ternaire (elle n'a jamais été réduite). Montrons par récurrence qu'un littéral ne peut avoir qu'une seule occurence parmi l'ensemble des littéraux des feuilles de T. Au début, T ne contenait qu'une seule clause, dont tous les littéraux étaient différents. Ensuite, à chaque amincissement, au pire une clause binaire $y \vee \overline{l}$ était insérée une seule fois car seule une feuille de T pouvait contenir l. Après insertion, il y a une seule occurence de y (nouvelle variable) et une seule occurence de $\overline{l}$ car il n'y en avait pas avant (sinon, il aurait fallu qu'il y ait une deuxième occurence de l dans une clause qui aurait fait disparaître $\overline{l}$ par résolution, car la racine de T ne contient pas $\overline{l}$).*
  Comme un littéral ne peut avoir qu'une seule occurence parmi l'ensemble des littéraux des feuilles de T, il ne peut y avoir au pire qu'une insertion de la clause binaire $x \vee \overline{l_1}$ dans T.
- *La clause de la racine de T est binaire (elle a*

été réduite une fois). Avant d'être réduite, chaque littéral de feuille n'avait qu'une occurence. Lorsqu'elle a été réduite par une insertion de $y \vee \overline{l}$ et une insertion de $y \vee \overline{l'}$, un seul littéral se trouve alors en double dans les feuilles de T : y. A partir de là, à chaque amincissement, il y a eu au pire deux insertions dans T, lorsque le renommage s'est fait sur y ou un de ses "renommeurs". Notons le fait que renommer y (ou un de ses "renommeurs") dans le sous-arbre de renommage l'empêche d'être renommé plus tard dans ce même sous-arbre car il a disparu de la racine donc il n'est plus source de généalogie. Donc, si l_1 est un des "renommeurs" de y, $x \vee \overline{l_1}$ est inséré deux fois dans T et x remplace l_1 dans la racine de T.

– *La clause de la racine de T est unaire (elle a été réduite deux fois). Avant d'être une deuxième fois réduite, chaque littéral de feuille avait au maximum deux occurences. Lorsqu'elle a été réduite une deuxième fois par une insertion de $y \vee \overline{l}$ et une insertion de $y \vee \overline{l'}$, la racine de T était $l \vee l'$. Au pire, l avait deux occurences dans les feuilles de T et l' une seule (ou l'inverse). En effet, l était un "renommeur" de la variable ayant été ajoutée deux fois lors de la première réduction de la racine de T contrairement à l' (sinon il y aurait eu deux réductions). Il y a donc eu en tout au maximum deux insertions de $y \vee \overline{l}$ et une insertion de $y \vee \overline{l'}$ dans T, lors de la deuxième réduction. Seul le littéral y a trois occurences dans les feuilles de T. Ensuite, à chaque amincissement, trois clauses (ou aucune) sont insérées dans T et la clause unaire de la racine est remplacée par un "renommeur" de y. Donc, dans notre cas, la racine de T est la clause l_1, la clause $x \vee \overline{l_1}$ est insérée trois fois dans T et la racine de T devient x.*

Puisqu'à chaque amincissement, une clause ternaire est inséré dans l'arbre de preuve et qu'au maximum 3 clauses binaires sont insérés dans chaque sous-arbre de renommage, qu'il y a moins de L sous-arbres de renommage et moins de L amincissements, à la fin de la série d'amincissements qui permet à l'arbre de preuve d'avoir une largeur 3, la nouvelle longueur de la preuve sera inférieure à (3L+1)L.

Une conséquence est que si une preuve arborescente est de longueur polynômiale alors elle reste polynômiale après être amincie jusqu'à avoir 3 de largeur. Ceci va nous aider à montrer qu'il existe une preuve polynômiale du problème des pigeons dans ER3.

6 Preuve polynômiale pour les pigeons

Tout d'abord, nous allons rappeler la preuve polynômiale dans ER du problème des pigeons qu'avait donné Cook [5] (sans qu'il ne l'ait complètement explicité).

Le problème des pigeons PHP(n) consiste à mettre n pigeons dans $n-1$ trous. P_{ij} signifie "le pigeon i est dans le trou j". La liste des clauses est :

– $\forall i,\ 1 \leq i \leq n, : P_{i1} \vee \ldots \vee P_{i,n-1}$
– $\forall i, j, k,\ 1 \leq k \leq n-1,\ 1 \leq i < j \leq n : \overline{P_{ik}} \vee \overline{P_{jk}}$

Voici le principe de la preuve par résolution étendue tel qu'il apparaît dans l'article de Cook. On va exprimer la satisfiabilité de PHP(n) en fonction de la satisfiabilité de PHP($n-1$). L'idée est qu'on peut obtenir une solution de PHP($n-1$) à partir d'une solution de PHP(n) en gardant les pigeons dans les mêmes trous à part que le pigeon qui se trouvait dans le trou $n-1$ (qui n'existe pas dans PHP($n-1$)) prend la place du pigeon n (qui n'existe pas dans PHP($n-1$)). Q_{ij} signifie "le pigeon i est dans le trou j" pour PHP($n-1$). On a donc les relations suivantes entre les P_{ij} et les Q_{ij} : $\forall i, j,\ 1 \leq i \leq n-1,\ 1 \leq j \leq n-2$: $Q_{ij} \Leftrightarrow (P_{ij} \vee (P_{i,n-1} \wedge P_{nj}))$. Remarquons que cette extension ternaire peut être exprimée grâce à deux extensions binaires disjonctives : $Q_{ij} \Leftrightarrow (P_{ij} \vee \overline{R_{ij}})$ et $R_{ij} \Leftrightarrow (\overline{P_{i,n-1}} \vee \overline{P_{nj}})$ mais nous garderons la formulation de Cook par simplicité. Sous forme CNF, on obtient donc les clauses :
(1) $Q_{ij} \vee \overline{P_{ij}}$,
(2) $Q_{ij} \vee \overline{P_{i,n-1}} \vee \overline{P_{nj}}$,
(3) $\overline{Q_{ij}} \vee P_{ij} \vee P_{i,n-1}$ et
(4) $\overline{Q_{ij}} \vee P_{ij} \vee P_{nj}$.
Ce sont les clauses d'extensions qu'on ajoute à PHP(n). Il s'avère qu'en effectuant un nombre polynômial (en $\Theta(n^3)$) de résolutions, on génère les clauses CNF de PHP($n-1$) à partir de celles de PHP(n) et des clauses d'extensions. On se ramène donc en un temps polynômial de PHP(n) à PHP($n-1$). Il suffit de réitérer le processus de manière récursive (ajouter de nouvelles clauses d'extensions pour se ramener à chaque fois à un problème de taille juste inférieur) pour aboutir à la génération (en temps polynômial) des clauses de PHP(2) qui ne nécessitent que 3 résolutions pour obtenir la clause vide.

Voici maintenant le détail de la résolution avec clauses d'extensions non explicités dans l'article de Cook.

La figure 4 indique comment on obtient chaque clause $\overline{Q_{ik}} \vee \overline{Q_{jk}}$ en sept résolutions.

Voici comment on obtient chaque clause $Q_{i1} \vee \ldots \vee Q_{i,n-2}$ en $2n-2$ résolutions. On note C_j la clause $Q_{i1} \vee \ldots \vee Q_{ij} \vee P_{i,j+1} \vee \ldots \vee P_{i,n-2}$ et D_j la clause $Q_{i1} \vee \ldots \vee Q_{ij} \vee P_{n,j+1} \vee \ldots \vee P_{n,n-2} \vee \overline{P_{i,n-1}}$. $\forall j$, C_j se génère par résolution entre C_{j-1} et $Q_{ij} \vee \overline{P_{ij}}$.

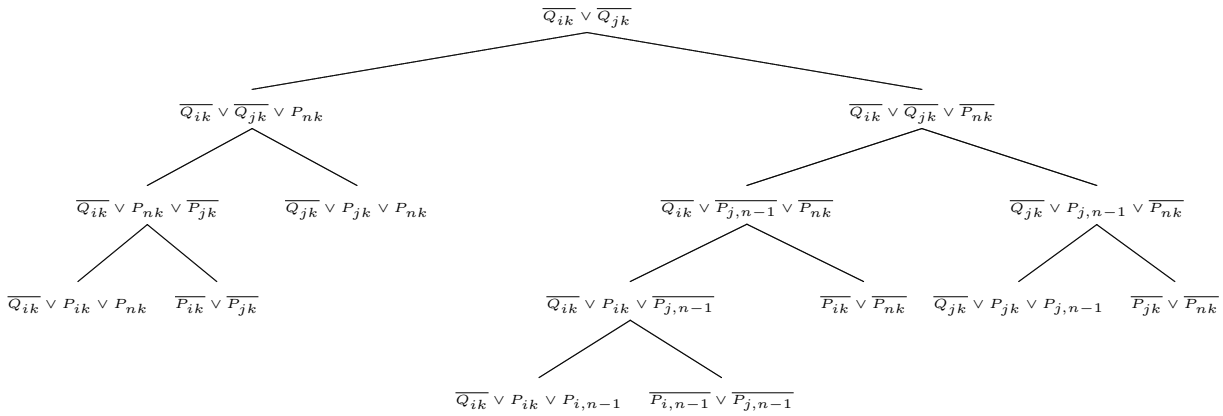


FIG. 4

Itérativement, on part de $P_{i1} \vee \ldots \vee P_{i,n-1} = C_0$ et on finit par générer $C_{n-2} = Q_{i1} \vee \ldots \vee Q_{i,n-2} \vee P_{i,n-1}$. $\forall j$, D_j se génère par résolution entre D_{j-1} et $Q_{ij} \vee \overline{P_{i,n-1}} \vee \overline{P_{nj}}$. Par résolution entre $P_{n1} \vee \ldots \vee P_{n,n-1}$ et $\overline{P_{i,n-1}} \vee \overline{P_{n,n-1}}$, on obtient D_0. Puis, itérativement, on finit par générer $D_{n-2} = Q_{i1} \vee \ldots \vee Q_{i,n-2} \vee \overline{P_{i,n-1}}$. Enfin, par résolution entre C_{n-2} et D_{n-2}, on obtient $Q_{i1} \vee \ldots \vee Q_{i,n-2}$.

La version 3-SAT du problème des pigeons se différencie uniquement par ses clauses n-aires positives $P_{i,1} \vee \ldots \vee P_{i,n-1}$ qui sont remplacées par $n-3$ clauses ternaires $P_{i,1} \vee P_{i,2} \vee \overline{X_{i,2}}$, $X_{i,2} \vee P_{i,3} \vee \overline{X_{i,3}}$, ..., $X_{i,n-4} \vee P_{i,n-3} \vee \overline{X_{i,n-3}}$ et $X_{i,n-3} \vee P_{i,n-2} \vee P_{i,n-1}$.

Proposition 5 *La version 3-SAT du problème des pigeons se résout en temps polynômial dans* ER3.

Preuve 5 *D'abord, on peut remarquer que la preuve polynômiale de Cook est une composition de dérivations arborescentes. Les seules dérivations arborescentes qui ne sont pas de largeur 3 sont celles qui permettent de dériver les clauses $q_i = Q_{i1} \vee \ldots \vee Q_{i,n-2}$ à partir des clauses $p_i = P_{i1} \vee \ldots \vee P_{i,n-1}$ et $p_n = P_{n1} \vee \ldots \vee P_{n,n-1}$ (et d'autres clauses, binaires ou ternaires). Pour obtenir une preuve de largeur 3, il nous suffit donc de trouver les dérivations arborescentes qui vont dériver les clauses ternaires équivalentes à q_i ($Q_{i,1} \vee Q_{i,2} \vee \overline{Y_{i,2}}$, $Y_{i,2} \vee Q_{i,3} \vee \overline{Y_{i,3}}$, ..., $Y_{i,n-5} \vee Q_{i,n-4} \vee \overline{Y_{i,n-4}}$ et $Y_{i,n-4} \vee Q_{i,n-3} \vee Q_{i,n-2}$) à partir des clauses ternaires équivalentes à p_i, p_n et les mêmes autres clauses binaires et ternaires qu'auparavant. Pour ceci, nous construisons d'abord des dérivations de largeur non borné. Pour obtenir les clauses ternaires équivalentes à q_i à partir des clauses ternaires équivalentes à p_i et p_n :*

- *nous dérivons p_i et p_n (nous les reconstituons) à partir de leurs clauses ternaires équivalentes (en $n-2$ résolutions chacune),*
- *puis nous dérivons q_i comme auparavant,*
- *puis à partir de q_i et des extensions $Y_{i,2} \Leftrightarrow (Q_{i,1} \vee Q_{i,2})$ et $\forall j, 3 \leq j < n-4$, $Y_{i,j} \Leftrightarrow (Q_{i,j-1} \vee Y_{i,j-1})$, nous dérivons linéairement (en $2n-6$ résolutions) $Y_{i,n-4} \vee Q_{i,n-3} \vee Q_{i,n-2}$.*

Les autres clauses que nous voulons ($Q_{i,1} \vee Q_{i,2} \vee \overline{Y_{i,2}}$, $Y_{i,2} \vee Q_{i,3} \vee \overline{Y_{i,3}}$, ..., $Y_{i,n-5} \vee Q_{i,n-4} \vee \overline{Y_{i,n-4}}$) sont celles données par les extensions. Nous avons donc les dérivations arborescentes de longueur polynômiale que nous voulions, à part que leur largeur est en général supérieure à 3. Mais comme nous avons vu (proposition 4) que nous pouvons les amincir jusqu'à ce qu'elles soient de largeur 3 en n'augmentant leur longueur que de manière polynômiale, nous pouvons remplacer toutes les dérivations arborescentes trop larges par leur version amincie de largeur 3. Nous pouvons en conclure qu'il existe bien une réfutation polynômiale du problème des pigeons dans ER3.

Le cas du problème des pigeons peut se généraliser à tout problème exponentiel dans Res, polynômial dans ER et dont la dérivation dans ER peut se décomposer en sous-dérivations arborescentes dont les racines et les feuilles contiennent toutes des clauses de taille 3 maximum. Par la même technique d'amincissements successifs de chaque sous-dérivation arborescente, on peut obtenir une dérivation globale de largeur 3 de longueur polynômiale.

Il existe donc des problèmes dont la réfutation est de longueur polynômiale dans ER3 alors qu'elle est exponentielle dans Res. Cela signifie que la résolution étendue, même si on restreint la largeur de ses preuves, peut rester plus efficace sur certains problèmes que la résolution standard. Il n'est évidemment pas exclu qu'il existe d'autres problèmes pour lesquels le phénomène inverse soit vrai mais cela légitime qu'on considère ER3 comme une alternative possible à Res.

Un schéma général d'algorithme pourrait être le suivant. En partant d'un ensemble B contenant toutes les clauses d'entrée d'une formule CNF, on complète B avec des résolvantes de taille 3 maximum jusqu'à saturation. Si la clause vide n'a pas été trouvée alors on ajoute à B les 3 clauses d'une extension binaire disjonctive non redondante et on recommence. Cet algorithme est complet car si la formule est satisfiable, il y a aura un moment où on ne pourra plus ajouter d'extensions non redondantes (on aura ajouté toutes les extensions possibles de niveau inférieur à $n \log n$) et on pourra alors conclure à la satisfiabilité de la formule.

Chaque phase de saturation par résolution peut produire de l'ordre de $\Theta((n+n')^3)$ résolvantes où n est le nombre de variables d'entrée et n' est le nombre actuel de variables d'extension. Celà est à comparer avec les $\Theta((n+n')^n)$ résolvantes potentielles qui pourraient être produites si on ne limitait pas la taille des résolvantes. Evidemment, n' n'est pas nécessairement borné par une fonction polynômiale. Par contre, si une formule n'a besoin qu'on lui ajoute qu'un nombre polynômialement borné d'extensions, cet algorithme est susceptible de lui trouver une preuve de longueur polynômiale en un temps polynômial, du moment que les bons choix d'extensions sont faits.

Conclusion et perspectives

En démontrant que l'on pouvait restreindre la résolution étendue à la dérivation de résolvantes ternaires tout en restant complet, nous avons diminué très fortement la combinatoire de ce système de preuve. En montrant que dans ce cadre le problème des pigeons reste réfutable en temps polynômial, nous avons établi que la résolution étendue de largeur 3 est une alternative possible à la résolution standard pour un certain nombre de problèmes dont la quantité reste à définir. Pour l'instant, une question théorique reste ouverte : `ER3` est-elle aussi puissante que `ER`, i.e. lorsque `ER` est capable de trouver une preuve de longueur polynômiale à un problème, `ER3` le peut-il toujours aussi ? En effet, le résultat de polynômialité sur la longueur des preuves amincies ne concerne que les arbres et pas toutes les preuves sous forme de DAG[1].

Dans cet article, nous n'avons pas abordé d'aspect pratique. Si on veut que la résolution étendue de largeur 3 soit efficace, il reste à trouver une bonne heuristique de choix d'extension dans ce contexte. C'est cet aspect qu'il faudra aussi maintenant explorer.

[1]Depuis la soumission de cet article, nous avons établi que `ER3` est aussi puissante que (i.e., p-simule) `ER`. Ce résultat n'étant pas encore formellement validé par la communauté suite à une procédure de relecture, nous ne l'avons pas intégré à la version finale de cet article.

Remerciements

Ce travail a été soutenu par un programme blanc de l'ANR (projet UNLOC).

Références

[1] Gilles Audemard, George Katsirelos, and Laurent Simon. A restriction of extended resolution for clause learning sat solvers. In *AAAI*, 2010.

[2] Gilles Audemard and Laurent Simon. Gunsat : A greedy local search algorithm for unsatisfiability. In *IJCAI*, pages 2256–2261, 2007.

[3] Eli Ben-Sasson and Avi Wigderson. Short proofs are narrow - resolution made simple. *J. ACM*, 48(2) :149–169, 2001.

[4] Vasek Chvátal and Endre Szemerédi. Many hard examples for resolution. *J. ACM*, 35(4) :759–768, 1988.

[5] Stephen A. Cook. A short proof of the pigeon hole principle using extended resolution. *SIGACT News*, 8 :28–32, 1976.

[6] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *J. ACM*, 7(3) :201–215, 1960.

[7] Armin Haken. The intractability of resolution. *Theor. Comput. Sci.*, 39 :297–308, 1985.

[8] Jinbo Huang. Extended clause learning. *Artif. Intell.*, 174(15) :1277–1284, 2010.

[9] Robert A. Kowalski and Donald Kuehner. Linear resolution with selection function. *Artif. Intell.*, 2(3/4) :227–260, 1971.

[10] Chu Min Li and Anbulagan. Heuristics based on unit propagation for satisfiability problems. In *IJCAI (1)*, pages 366–371, 1997.

[11] John Alan Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1) :23–41, 1965.

[12] G. S. Tseitin. On the complexity of derivation in propositional calculus. In *Studies in Constructive Mathematics and Mathematical Logics*, pages 115–125, 1968.

[13] Alasdair Urquhart. Hard examples for resolution. *J. ACM*, 34(1) :209–219, 1987.

Amélioration de l'expressivité des contraintes de table

Jean-Charles Régin

Université de Nice-Sophia Antipolis, I3S CNRS
2000, route des Lucioles - Les Algorithmes - BP 121
06903 Sophia Antipolis Cedex - France
jcregin@gmail.com

La complexité des algorithmes de cohérence d'arc pour les contraintes exprimées en extension dépend principalement du nombre de tuples impliqués. Aussi, plusieurs travaux ont été menés à bien afin de diminuer ce nombre. Notamment, Katsirelos et Walsh ont proposé de compresser les tuples. Ils ont intégré leur méthode dans GAC-Schema et montré comment on peut représenter un ensemble de F tuples interdits par un ensemble de ndF tuples compressés. Enfin, ils ont laissé ouvert la question de l'intégration des tuples compressés avec les meilleures versions de GAC-Schema comme celle de Lhomme et Régin. Dans ce papier, nous introduisons une généralisation des tuples compressés : les séquences de tuples. Nous montrons comment les meilleurs versions de GAC-Schema peuvent être aisément adaptées pour utiliser ces nouveaux tuples et comment F tuples interdits peuvent être facilement représentés par $F + 1$ tuples compressés, ce qui élimine tout besoin d'un algorithme particulier dans le cas de tuples interdits. Nous mentionnons aussi l'intérêt de notre approche au niveau de l'expressivité des contraintes de table.

1 Introduction

Les contraintes de table (Table constraints) font parties des contraintes les plus utilisées en PPC. Elles peuvent être définies à partir de la liste des combinaisons de valeurs autorisées (tuples autorisés), par la liste des tuples interdits ou par une fonction Booléenne équivalente à la satisfaction de la contrainte.

Un algorithme générique permettant de calculer la fermeture par arc consistance de ces contraintes a été développé. Il s'agit de GAC-Schema [1]. Il est basé sur la traversée de la liste des tuples autorisés jusqu'à en trouver un valide, c'est-à-dire tel que chaque valeur du tuple appartienne au domaine courant de sa variable. Sa complexité depend donc du nombre de tuples autorisés. Aussi, certains travaux récents ont été menés à bien afin de restructurer l'ensemble des tuples pour en diminuer le nombre [4, 5, 6, 2]. En particulier, Katsirelos et Walsh [5] ont proposé de compresser les tuples en utilisant des tuples "Glocal Cut Seed" (tuples GCS [3]). Un tel ensemble de tuples contient tous les tuples du produit Cartésien de sous-ensembles des domaines des variables. Par exemple, le tuple compressé $(\{1,2\},\{2\},\{1,2\})$ est équivalent à l'ensemble des tuples $\{(1,2,1),(1,2,2),(2,2,1),(2,2,2)\}$. En compressant la liste des tuples autorisés ils ont montré que la fermeture par arc consistance pouvait être accélérée. En outre, ils ont expliqué comment leur algorithme pouvait être intégré dans GAC-Schema.

Les travaux de Katsirelos et Walsh sont principalement focalisé sur l'algorithme calculant cette compression. Ils ne se sont pas réellement intéresser à l'expressivité de ces tuples compressés. Il sont seulement considérés ce point afin de représenter les tuples interdits à l'aide de tuples autorisés. Malheureusement, la méthode qu'ils proposent peut requérir nd fois plus de tuples compressés qu'il n'y a de tuples interdits.

Dans ce papier, nous proposons de montrer comment une simple généralisation des tuples GCS, que nous appellerons séquence de tuples, est beaucoup plus intéressante pour représenter des groupes de tuples. Une séquence de tuples est constituée d'un tuple GCS associé à un tuple minimum et un tuple maximum (pris par rapport à un ordre). L'avantage de ces ensembles de tuples est qu'il permettent de représenter facilement les tuples interdits car on peut grâce

à eux représenter facilement le complémentaire d'un ensemble.

Par exemple, considérons une contrainte de table définie sur des variables ayant pour domaines l'ensemble $\{a, b, c, d\}$ et dont la liste des tuples interdits est (a, b, c, d), (b, c, d, a) et (d, d, a, a). Pour simplifier, les tuples sont ordonnés de façon lexicographique. Nous pouvons définir une séquence de tuples qui représente les tuples du tuple (a, a, a, a) au tuple (a, b, c, c) qui est le tuple qui précède immédiatement le tuple (a, b, c, d). Cette séquence de tuple est simplement représentée par le tuple minimum $min = (a, a, a, a)$, le tuple maximum $max = (a, b, c, c)$ et le tuple GCS $(\{a, b, c, d\}, \{a, b, c, d\}, \{a, b, c, d\}, \{a, b, c, d\})$. Nous écrirons cette séquence : $((a, a, a, a), (a, b, c, c), (\{a, b, c, d\}, \{a, b, c, d\},$ $\{a, b, c, d\}, \{a, b, c, d\}))$. Bien sûr, nous pourrions aussi ne pas introduire d'autres valeurs que a pour la première variable, mais cela n'est même pas nécessaire. Ensuite, nous pouvons répéter cette méthode et définir la séquence de tuples $((a, b, d, a), (b, c, c, d), (\{a, b, c, d\}, \{a, b, c, d\}, \{a, b, c, d\},$ $\{a, b, c, d\}))$ représentant les tuples de (a, b, d, a) à (b, c, c, d) et la séquence de tuples qui représente les tuples de (b, c, d, b) à (d, c, d, d) et enfin une séquence de tuples correspondant aux tuples de (d, d, a, b) à (d, d, d, d). Ainsi, nous pouvons représenter F tuples interdits avec seulement $F + 1$ séquences de tuples. C'est une amélioration de la méthode de Katsirelos et Walsh par un facteur nd.

Nous allons aussi montrer que les séquences de tuples peuvent être utiles pour améliorer l'expressivité des contraintes de tables, parce qu'elle permettent de mélanger des tuples interdits et des tuples autorisés.

`Organisation de l'article.` Tout d'abord, nous rappellerons certaines définitions. Puis, nous introduirons la notion de cluster de tuples et nous définirons les séquences de tuples. Ensuite, nous expliquerons comment représenter un ensemble de tuples interdits par des séquences de tuples autorisés et nous montrons comment les séquences de tuples peuvent améliorer l'expressivité des contraintes de table. Avant de conclure, nous détaillerons l'intégration de séquences de tuples dans GAC-Schema.

2 Préliminaires

Réseau de contraintes. Un *réseau de contraintes fini* $\mathcal{N} = (X, \mathcal{D}, \mathcal{C})$ est défini par un ensemble de n *variables* $X = \{x_1, \ldots, x_n\}$, un ensemble de *domaines* $\mathcal{D} = \{D(x_1), \ldots, D(x_n)\}$ où $D(x_i)$ est l'ensemble fini des *valeurs* possibles pour la variable x_i, et un ensemble $\mathcal{C}$ de *contraintes* entre variables. La valeur a d'une variable x est souvent notée (x, a).

Contrainte. Une contrainte C portant sur l'ensemble ordonné de variable $X(C) = (x_{i_1}, \ldots, x_{i_r})$ est un sous-ensemble $T(C)$ du produit Cartésien $D(x_{i_1}) \times \cdots \times D(x_{i_r})$ qui spécifie les combinaisons *autorisées* de valeurs pour les variables $x_{i_1}, \ldots, x_{i_r}$. Un élément de $D(x_{i_1}) \times \cdots \times D(x_{i_r})$ est appelé un *tuple* sur $X(C)$ et $\tau[x]$ est la valeur de τ affectée à x. Les tuples sur $X(C)$ non autorisés par C sont appelés des tuples *interdits* de C. $|X(C)|$ est *l'arité* de C. Sans perte de généralité nous pouvons considérer que les tuples autorisés sont ordonnés par la relation $\prec$. Nous noterons d la taille du plus grand domaine initial et n l'arité.

Arc consistance. Soit C une contrainte. Un tuple τ sur $X(C)$ est *valide* si et seulement si $\forall x \in X(C), \tau[x] \in D(x)$; et τ est un *support* pour (x, a) ssi $\tau[x] = a$ et $\tau \in T(C)$. Une valeur $a \in D(x)$ est *consistante avec* C ssi $x \notin X(C)$ ou s'il existe un support valide pour (x, a). C est *arc consistant* ssi $\forall x \in X(C), D(x) \neq \emptyset$ et $\forall a \in D(x)$, a est consistant avec C.

Ordre Lexicographique. Un ordre total $<_d$ peut être défini sur $D(x), \forall x \in X$, sans perte de généralité. Deux tuples τ et τ' sur $X(C)$ peuvent être ordonnés par l'ordre lexicographique naturel $\prec_{lo}$ dans lequel $\tau \prec_{lo} \tau'$ ssi $\exists k$ tel que $\forall j \in [1..k-1], \tau[x_{i_j}] = \tau'[x_{i_j}]$ et $\tau[x_{i_k}] <_d \tau'[x_{i_k}]$.

3 Cluster de tuples

Un cluster de tuples est un concept général qui encapsule le concept de groupe de tuples à condition que l'on puisse les utiliser efficacement avec GAC-Schema.

Definition 1 *Un* **cluster de tuples** *est une structure de données correspondant à un ensemble de tuples associé à une opération efficace retournant un tuple valide de cet ensemble s'il existe et nil sinon.*

Les tuples "Global Cut Seed" (tuples GCS) proposés par Focacci et Milano [3] sont un exemple de cluster de tuples. Un tuple GCS est équivalent à l'ensemble des tuples obtenus en calculant le produit Cartésien de sous-ensembles des domaines des variables. Par exemple, le tuple compressé $(\{a, b\}, \{b, c\}, \{a, d\})$ est équivalent à l'ensemble de tuples $\{(a, b, a), (a, b, d), (a, c, a), (a, c, d), (b, b, a), (b, b, d),$ $(b, c, a), (b, c, d)\}$.

Il est facile de vérifier si un tuple GCS contient un tuple valide ou pas : s'il n'y a pas d'intersection entre le domaine du tuple GCS et le domaine courant correspondant, alors au moins un tuple valide est contenu dans le tuple GCS. Il est aussi facile de vérifier si un tuple GCS contient un support pour la valeur (x, a) : on fixe le domaine de x à $\{a\}$ et on calcule les intersections entre les domaines du tuple GCS et ceux des va-

riables. S'il n'y a pas d'intersection vide alors le tuple GCS contient un tuple valide impliquant (x, a), c'est-à-dire un support pour (x, a). Cette opération peut être effectuée en $O(nd)$ dans le pire des cas, car tester si l'intersection de deux domaines n'est pas vide peut être fait en $O(d)$, pour des domaines de taille d. Nous verrons plus tard comment améliorer ce point.

Dans cet article, nous proposons une généralisation de tuples GCS : les séquences de tuples. Dans un tuple GCS il n'y a pas d'ordre et nous avons quelques problèmes pour exprimer certains sous-ensembles de tuples obtenus par les produits Cartésiens des domaines. Les séquences de tuples permettent de remédier à cet inconvénient en introduisant des bornes à cette énumération. Une séquence de tuples est un tuple GCS associé à un tuple minimum et un tuple maximum respectant l'ordre lexicographique. Autrement dit, il contient tous les tuples d'un tuple GCS qui sont plus grands ou égaux que le tuple minimum et plus petits ou égaux que le tuple maximum. Formellement, on a :

Definition 2 *Soit C une contrainte de table définie sur $X(C) = \{x_1, ..., x_n\}$.*
Une **séquence de tuples** *s définie par le triplet $(\tau^s_{min}, \tau^s_{max}, (D^s(x_1), D^s(x_2), ..., D^s(x_n)))$ est l'ensemble des tuples t tels que $t \in D^s(x_1) \times \cdots \times D^s(x_n)$ et $\tau^s_{min} \preceq_{lo} t \preceq_{lo} \tau^s_{max}$*

L'avantage principal des séquences de tuples est leur capacité à représenter de façon positive un ensemble de tuples interdits. Même s'il est possible de compresser les domaines, par exemple un seul joker ('*') peut représenter toutes les valeurs du domaine initial, nous devons considérer que la consommation mémoire d'une séquence de tuples est égale au nombre de valeurs qu'elle contient. Le minimum est n et le maximum est en $O(nd)$. Dans ce cas, il est important de rappeler qu'un nombre exponentiel de tuples peuvent être représenté par une seule séquence de tuples.

4 Tuples interdits

Nous allons montrer comment un ensemble de tuples interdits peut-être représenté par un ensemble de séquences de tuples autorisés. L'idée est de représenter cet ensemble par son complémentaire. Puis, nous verrons comment cet ensemble complémentaire peut être représenté par des séquences de tuples autorisés.

Nous utiliserons les notations suivantes :

- la valeur minimale de $D(x)$ est notée $\underline{D}(x)$, et sa valeur maximale est notée $\overline{D}(x)$.
- $\underline{\tau}(C)$ est le nombre minimum de combinaisons de valeurs sur $X(C)$: $(\underline{D}(x_1), ..., \underline{D}(x_n))$
- $\overline{\tau}(C)$ est le nombre maximal de combinaisons de valeurs sur $X(C)$: $(\overline{D}(x_1), ..., \overline{D}(x_n))$
- *n-comb*(t, C) est la combinaison de valeurs suivant immédiatement t pour $X(C)$ par rapport à l'ordre lexicographique. Elle est égale à *nil* si t est la dernière combinaison.
- *p-comb*(t, C) est la combinaison de valeurs précédant immédiatement t pour $X(C)$ par rapport à l'ordre lexicographique. Elle est égale à *nil* si t est la première combinaison.

Considérons une contraint de table C définie sur $X(C) = \{x_1, ..., x_n\}$ dont le domaine de x_i est $D(x_i)$, et $F = \{f_1, ...f_q\}$ un ensemble de tuples interdits ordonnés lexicographiquement. Soit$P$ le produit Cartésien des domaines des variables de $X(C)$, c'est-à-dire $D(x_1) \times \cdots \times D(x_n)$. Alors, l'ensemble des tuples autorisés est égal à $A = P - F$.

Soit $\mathcal{D}$ le tuple GCS constitué de tous les domaines des variables de $X(C)$. L'ensemble A peut être représenté par au plus $|F|+1$ séquences de tuples autorisés, comme suit :

- la première séquence de tuples est définie par ($\underline{\tau}(C)$,*p-comb*(f_1, C),$\mathcal{D}$).
- pour $i = 2, ..., n$ la séquence de tuples est définie par (*n-comb*(f_{i-1}, C),*p-comb*(f_i, C),$\mathcal{D}$)
- la dernière séquence de tuples est (*n-comb*$(f_n$,$\overline{\tau}(C)$,$\mathcal{D}$)

Bien entendu, la partie GCS de ces séquences de tuples peut être raffinée. On remarquera que certains tuples peuvent aussi être vides. C'est le cas lorsque le tuple minimum ou maximum d'une séquence est *nil* ou lorsque le tuple maximum est plus petit que le tuple minimum. Il n'est pas nécessaire de représenter de telles séquences.

Clairement, chaque séquence de tuples peut être représentée en $O(nd)$. Aussi, la consommation globale en mémoire est en $O(nd|F|)$. C'est supérieur à la représentation de $|F|$ tuples interdits qui est en $O(n|F|)$ (car un tuple contient n valeurs), mais c'est moins que la méthode de Katsirelos et Walsh qui requiert $O(nd|F|)$ tuples GCS et qui a donc une complexité mémoire en $O(n^2d^2|F|)$. En conclusion, nous gagnons un facteur nd par rapport à une représentation à l'aide tuples GCS et nous perdons un facteur d par rapport à la représentation explicite des tuples interdits.

Un autre avantage de notre approche par rapport à celle de Katsirelos and Walsh est sa grande simplicité.

5 Expressivité

L'idée principale des séquences de tuples est d'introduire une certaine structure dans la définition de

contraintes de table. Les contraintes de table sont fréquemment utilisées parce qu'elle peuvent exprimer n'importe quel type de contraintes, par exemple des règles entre des variables. En outre, la fermeture par arc consistance peut être calculée, même si cela peut s'avérer coûteux. En conséquence, grâce aux contraintes de table, nous pouvons exprimer des contraintes complexes et bénéficier d'algorithmes de filtrage associés puissants. L'inconvénient majeur est que le nombre de tuples définissant certaines contraintes de table peut-être tellement grand que l'on ne peut pas définir la contrainte. Le but des clusters de tuples est justement de contourner cet inconvénient en réduisant le nombre de tuples nécessaire pour exprimer certaines contraintes en extension. Les clusters de tuples permettent en plus de pouvoir facilement exprimer certaines contraintes. Nous proposons de détailler un peu ce point.

Dans la pratique, les contraintes de table sont bien souvent définies de plusieurs façons [1] :

• les tuples autorisés proviennent d'une base de données.

• toutes les solutions d'un sous-problèmes sont calculées et elles forment l'ensemble des tuples autorisées.

• l'utilisateur veut exprimer directement certaines compatibilités et incompatibilités entre des combinaisons de valeurs. Dans ce cas, il est utile de pouvoir exprimer que certaines combinaisons de valeurs sont autorisées alors que d'autres sont interdites. Par exemple, (x_1, a) ne peut être combiné qu'avec (x_2, b).

Dans le premier cas, les méthodes de compression sont certainement les meilleures méthodes pour représenter ces tuples. On peut aussi les utiliser pour le second point, mais il peut être plus intéressant de calculer l'ensemble des solutions de façon à pouvoir compresser cet ensemble. En d'autres termes, la méthode d'énumération devrait essayer de compresser la tuples pendant le calcul de toutes les solutions, par exemple sous la forme de séquences de tuples.

Nous proposons détudier plus précisément le dernier cas.

Comme nous l'avons vu, les tuples GCS peuvent être utilisés pour représenter certaines combinaisons de valeurs. Cependant, les séquences de tuples ont une expressivité plus forte car elles permettent de représenter efficacement des contraintes définies à la fois par des tuples autorisés et par des tuples interdits comme nous allons le montrer.

Considérons un ensemble A de séquences de tuples autorisés et un ensemble F de tuples interdits. Tout d'abord, on ordonne lexicographiquement l'ensemble des tuples interdits. Ensuite nous sélectionnons chaque séquence de tuples $s \in A$ et nous recherchons les tuples interdits qui sont contenus dans s. S'il y a k tuples interdits qui sont inclus dans s alors nous pouvons décomposer la séquence de tuples s en $k+1$ séquences de tuples avec une méthode similaire à celle que nous avons présentée dans la section précédante. Globalement, nous obtiendrons donc $|A| \times |F|$ séquences de tuples. Cette valeur est certainement un majorant du nombre de séquences réellement obtenues. Par exemple, si les séquences de tuples sont disjointes, ce qui est souvent le cas, alors nous aurons besoin de seulement $O(|A| + |F|)$ séquences, puisqu'un tuple interdit ne peut être contenu que dans une seule séquence de tuples autorisés.

Enfin, nous pourrions aussi considérer des séquences de tuples interdits, mais nous n'avons pas de méthode générale pour les combiner avec des séquences de tuples autorisés. Nous pouvons énumérer les tuples interdits et appliquer l'algorithme précédant et essayer de recombiner des séquences de tuples mais nous n'avons pas de garantie quant au nombre de tuples que nous obtiendrons à la fin. Ce problème mérite plus d'attention dans le futur.

6 Intégration dans GAC-Schema

Dans cette section, nous montrons comment les séquences de tuples peuvent être intégrées dans GAC-Schema. Tout d'abord nous montrons qu'une séquence de tuples est un cluster de tuples.

6.1 Tuple valide minimum

La recherche pour un tuple valide minimum [2] d'une séquence de tuples peut être effectuée en appliquant l'algorithme suivant :

1. On commence avec le tuple minimum de s (i.e. τ^s_{min}). Si τ^s_{min} est valide alors on le renvoie et on arrête l'algorithme. Sinon, on recherche le premier indice i tel que la valeur de τ^s_{min} impliquant x_i n'est plus dans le domaine de $D(x_i)$. C'est-à-dire que nous avons $\forall j = 1..i-1\ \tau^s_{min}[x_j] \in D(x_j)$ and $\tau^s_{min}[x_i] \notin D(x_i)$.
2. On recherche de i à 1, le premier indice j tel que $D^s(x_j)$ contienne une valeur valide plus grande que $\tau^s_{min}[x_j]$. Si un tel indice j n'existe pas alors on renvoie nil et l'algorithme s'arrête.
3. On crée un nouveau tuple t comme suit : pour $k = 1$ à $j-1$ on définit $t[x_k] = \tau^s_{min}[x_k]$; $t[x_j]$ contient

1. Il est aussi possible de définir une contrainte de table à partir d'un prédicat, mais cette méthode reste peu utilisée en pratique. Aussi, nous ne la considérerons pas.

2. le tuple valide minimum dans s est le tuple $t \in s$ tel que t est valide et il n'existe pas de tuple valide $t' \in s$ avec $t' \prec_{lo} t$.

la première valeur de $D^s(x_j)$ qui est valide et plus grande que $\tau^s_{min}[x_j]$ et pour $k = j+1$ à n on définit $t[x_k]$ comme la valeur minimum valide de $D^s(x_k)$. Si t est plus petit ou égal à τ^s_{max} alors l'algorithme renvoie t sinon il renvoie nil.

Par exemple, supposons que nous ayons $D(x_1) = \{a, b, c\}$,$D(x_2) = \{a\}$,$D(x_3) = \{a, b\}$ et $D(x_4) = \{a, b, c\}$ et $s = (t^s_{min} = (a, b, c, c), t^s_{max} = (c, b, b, b), (\{a, b, c\}, \{a, b, c\}, \{a, b, c\}, \{a, b, c\}))$. Alors, l'indice calculé par l'étape 1 est $i = 3$. D'après l'étape 2 nous calculons $j = 1$ et après l'étape 3 nous obtiendrons (b, a, a, a) qui est le premier tuple valide de s.

Clairement, cet algorithme est en $O(nd)$ parce que chaque étape est en $O(nd)$. Cela montre qu'une séquence de tuples est un cluster de tuples.

Il est aisé de rechercher le tuple valide minimum impliquant une valeur donnée (x, a). On a juste besoin de considérer que $D(x) = \{a\}$ dans l'algorithme. Aussi, nous pouvons facilement rechercher un nouveau support pour une valeur.

il est aussi intéressant de remarquer que cet algorithme peut être utilisé de façon incrémentale. Dans ce cas, nous pouvons prouver que la complexité en temps peut être amortie. Cet algorithme peut aussi être facilement modifiée pour recherche le tuple valide minimum plus grand qu'un autre tuple donné $\sigma \succeq \tau^s_{min}$: on a juste besoin de remplacer τ^s_{min} par σ dans l'algorithme. Donc, si pour une valeur (x, a) on doit effectuer plusieurs recherches, alors chaque recherche peut commencer là ou la précédante s'était arrêtée. De plus, il est clair que cela ne peut pas coûter plus que de traverser indépendamment toutes les tuples impliquant (x, a) et contenus dans la séquence. Cela signifie qu'utiliser des séquences de tuples ne peut pas coûter plus cher que de considérer indépendamment les tuples de ces séquences.

Nous pouvons maintenant détailler l'intégration dans GAC-Schema. Nous ne considérerons que la version de GAC-Schema qui traite les tuples autorisés.

6.2 Séquences de tuples au lieu de tuples

Calculer la fermeture par consistance d'arc signifie maintenir un support pour chaque valeur de chaque variable. Quand une valeur n'a plus de support alors elle peut être supprimée puisqu'elle n'est plus consistante avec la contrainte.

Il existe deux étapes dans les algorithmes généraux calculant la fermeture par consistance d'arc : une qui détermine les valeurs pour lesquelles un support doit être recherché ; et une autre qui recherche un nouveau support pour une valeur. La première étape est habituellement réalisée en associant à chaque valeur la liste de tuples contenant cette valeur et supportant une autre valeur. Ainsi, quand une valeur (x, a) est supprimée on connait les valeurs qui ont perdues leur support (parce que leur tuple support n'est plus valide) en traversant cette liste de tuples supportés. La seconde étape est la clé des algorithmes et les développements récents sont focalisés sur la rechercher d'un nouveau support pour une valeur donnée.

Détaillons ces deux étapes. Considérons qu'une valeur (x, a) a été supprimée de $D(x)$: pour toutes les valeurs (y, b) qui étaient supportées par un tuple contenant (x, a) un autre support doit être trouvé puisque le support courant n'est plus valide. Ces valeurs sont impliquées dans un tuple contenu dans une liste notée $S_C(x, a)$. GAC-Schema énumère tous les tuples dans les listes S_C et considère les valeurs valides supportées par ces tuples. Ensuite l'algorithme cherche un nouveau support pour elles.

La recherche d'un support valide est la tâche principale et la plus difficile de GAC-Schema. Par souci de clarté, nous ne parlerons pas dans cet article de l'aspect "multidirectionnel" de GAC-Schema et nous supposerons que la recherche d'un nouveau support est principalement basée sur l'étude des tuples autorisés de la contrainte. GAC-Schema considère successivement les tuples autorisés contenant la valeur pour laquelle on recherche un support jusqu'à en trouver un valide. Cela peut être fait efficacement en reliant entre eux les tuples autorisés contenant la même valeur. C'est-à-dire, que pour chaque valeur (y, b) nous définissons la liste des tuples autorisés l'impliquant. Cela coûte seulement un pointeur (vers le tuple suivant) pour chaque valeur de chaque tuple. C'est équivalent au coût mémoire pour représenter tous les tuples.

Il n'y a presque pas de changement si nous considérons des clusters de tuples au lieu de tuples : nous avons juste besoin de relier les deux notions. Quand nous considérons un tuple, nous avons aussi besoin de connaitre le cluster duquel il provient. L'ensemble de tuples devient un ensemble de clusters de tuples. Chaque valeur dans le cluster de tuples est reliée au prochain cluster de tuples qui la contient. Cela n'augmente pas la consommation mémoire puisqu'on ajoute seulement un pointeur pour chaque valeur de chaque cluster. Ainsi, lorsque l'on recherche un support pour (y, b), au lieu de traverser les tuples contenant (y, b), on traverse les clusters impliquant (y, b) en suivant les pointeurs associé à (y, b), jusqu'à ce qu'un tuple valide t appartenant au cluster courant soit trouvé. Un tel tuple valide est un support pour (y, b). Comme plusieurs tuples contenant (y, b) peuvent être valides dans le cluster courant on calcule le minimum en utilisant l'algorithme donné en section 6.1. Cette information est transmise à GAC-Schema ainsi que le cluster de tuples k d'où provient t. Autrement dit, une paire (t, k)

est renvoyée. Ensuite, GAC-Schema place t dans les listes S_C et utilise t et k pour calculer de nouveau supports.

Une paire (t, k) contient deux informations importantes qui vont s'avérer utiles pour contrôler la complexité en temps : k est le premier cluster de tuples contenant un tuple valide impliquant (y, b) et t est le tuple valide minimum contenant (y, b) dans k. Aussi, lorsque t ne sera plus valide on pourra continuer la recherche d'un support à partir de l'endroit où on s'était arrêté. On évite ainsi de répéter des calculs et on va pouvoir amortir la complexité en temps. De plus, on peut aller de cluster de tuples en clusters de tuples comme on le fait de tuples en tuples dans GAC-Schema. En conséquence, considérer des clusters de tuples au lieu de tuples peut économiser de la mémoire et n'augmente pas la complexité en temps dans le pire des cas. En pratique, cela améliore souvent l'algorithme.

Traverser la liste des tuples autorisés ou la liste des clusters de tuples autorisés contenant une valeur (y, b) lors de la recherche d'un support pour (y, b) est une méthode simple mais qui présente un inconvénient majeur : les domaines courants des variables ne sont pas considérés afin d'améliorer cette recherche. Les domaines sont utilisés uniquement pour tester la validité des tuples. En 2005, Lhomme et Régin [7] ont considérablement amélioré GAC-Schema en utilisant cette information sur les domaines pendant la recherche d'un nouveau support.

6.3 Intégration dans l'algorithme deLhomme et Régin

Dans GAC-Schema les tuples sont liés entre eux : chaque valeur est associée à un pointeur vers le prochain tuple la contenant. Lhomme et Régin ont montré que l'algorithme peut être accéléré si pour chaque tuple t et pour chaque valeur (x, a) on connait le premier tuple suivant t qui contient (x, a). Il est important de remarquer que nous avons besoin de cette information pour toutes les valeurs et pas seulement pour celles appartenant à t. Grâce à cette information, on va éviter de considérer de nombreux tuples de l'ensemble des tuples parce qu'on sait qu'ils ne peuvent pas être valides. En effet, le support pour une valeur (y, b) doit contenir (y, b) mais il doit aussi être valide, c'est-à-dire contenir des valeur valides pour les autres variables que y. La validité des valeurs est donc utilisée pendant la recherche de supports grâce à ce chainage.

Avec des tuples, il y a deux façons d'implémenter ce chainage :

• pour chaque tuple, on associe à chaque valeur de chaque variable, un pointeur qui représente le lien vers le prochain tuple contenant la valeur considérée. Cette méthode est particulièrement efficace en pratique. Cependant, elle multiplie la consommation mémoire par un facteur d ce qui est important !

• une structure de données complexe est utilisée : les "hologram tuples". Cette structure de données sacrifie une partie de la complexité en temps afin de réduire la complexité en espace. La consommation mémoire n'est pas augmentée mais on a besoin de $O(d)$ étapes pour accéder au pointeur suivant de n'importe quelle valeur.

Nous proposons d'appliquer les mêmes idées pour les clusters de tuples. Si on considère des clusters de tuples au lieu de tuples nous ne changerons pas l'algorithme. On associe à chaque valeur de chaque cluster de tuples un pointeur vers le prochain cluster de tuples contenant la valeur. Notons que si le cluster de tuple implique toutes les valeurs, nous pouvons atteindre le suivant de chaque valeur ! Si ce n'est pas le cas, alors nous pouvons ajouter cette information manquante, soit explicitement, ce qui augmente donc la consommation mémoire, soit en utilisant la structure de données des "hologram tuples" dans laquelle les tuples sont remplacés par des clusters. On remarquera que l'augmentation mémoire dans le premier cas est moins problématique avec les clusters de tuples qu'avec les tuples, parce qu'on dispose déjà de certaines informations dans les clusters puisqu'ils peuvent impliqués plus que n valeurs. Aussi, nous pouvons considérer qu'il n'y a pas de changement particulier si on utilise des clusters de tuples au lieu de tuples simples.

7 Discussion

Les séquences de tuples peuvent être généralisées en considérant pour chaque séquence de tuples un ordre spécifique pour les variables et un ordre spécifique pour les domaines de chaque variable. Cela ne changerait pas l'algorithme. Nous n'avons pas imposé de tels ordres parce que cela aurait complexifier les définitions et parce que nous n'avons pas trouvé d'exemple réel justifiant une telle modification.

Les techniques de compression, comme celle proposée par Katsirelos et Walsh pourrait aussi être utilisée dans notre cas, parce que les séquences de tuples intègre des tuples GCS. L'inconvénient de cette méthode est qu'elle peut être couteuse en temps (voir [5]). Néanmoins, il pourrait être intéressant d'examiner de plus près la possibilité de compresser des tuples dans des séquences de tuples au lieu de tuples GCS.

Nous avons aussi mentionné deux autres points qui méritent plus d'attention. Tout d'abord, quand la contrainte est définie à partir de l'ensemble des solutions d'un problème (habituellement définie par un

sous-ensemble de contraintes du problème général), on devrait essayer de compresser cet ensemble de solutions pendant qu'on les calcule. De cette façon, on pourrait améliorer le temps pour calculer toutes les solutions et obtenir un ensemble plus pertinent de tuples pour GAC-Schema. Ensuite, le problème de la combinaison d'ensembles de tuples interdits avec des ensembles de tuples autorisés devrait être considérée. Nous avons proposé une première méthode, mais elle est inefficace pour représenter certaines contraintes.

8 Conclusion

Dans cet article, nous avons introduit un nouveau type de cluster de tuples : les séquences de tuples. Ils généralisent les tuples GCS utilisés dans les contraintes de table par Katsirelos et Walsh. La représentation de tuples interdits par des séquences de tuples autorisés est simple et demande moins de mémoire que leur représentation par des tuples GCS. Il n'y a donc actuellement presque plus de raison pour utiliser une version dédiée de GAC-Schema pour les tuples interdits. Nous avons aussi montré que les séquences de tuples peuvent être utilisées pour mélanger des tuples autorisés et des tuples interdits de façon simple. Enfin, nous avons expliqué comment les clusters de tuples peuvent être facilement intégrés dans les meilleures implémentation de GAC-Schema comme celle de Lhomme et Régin.

Références

[1] C. Bessière and J-C. Régin. Arc consistency for general constraint networks : preliminary results. In *Proceedings of IJCAI'97*, pages 398–404, Nagoya, 1997.

[2] K. Cheng and R. Yap. An mdd-based generalized arc consistency algorithm for positive and negative table constraints and some global constraints. *Constraints*, 15, 2010.

[3] F. Focacci and M. Milano. Global cut framework for removing symmetries. In *Proc. CP'01*, pages 77–92, Paphos, Cyprus, 2001.

[4] I. Gent, C. Jefferson, I. Miguel, and P. Nightingale. Data structures for generalised arc consistency for extensional constraints. In *Proc. AAAI'07*, pages 191–197, Vancouver, Canada, 2007.

[5] G. Katsirelos and T. Walsh. A compression algorithm for large arity extensional constraints. In *Proc. CP'07*, pages 379–393, Providence, USA, 2007.

[6] C. Lecoutre. Optimization of simple tabular reduction for table constraints. In *Proc. CP'08*, pages 128–143, Sydney, Australia, 2008.

[7] O. Lhomme and J-C. Régin. A fast arc consistency algorithm for n-ary constraints. In *Proc. AAAI'05*, pages 405–410, Pittsburgh, USA, 2005.

CP-SST : approche basée sur la programmation par contraintes pour le test structurel du logiciel

Abdelilah Sakti **Yann-Gaël Guéhéneuc** **Gilles Pesant**

Département de Génie Informatique et Génie Logiciel
École Polytechnique de Montréal, Québec, Canada
{Abdelilah.Sakti, Yann-Gael.Gueheneuc, Gilles.Pesant}@polymtl.ca

Résumé

Le coût du test peut facilement dépasser 50% du coût total d'un logiciel critique. Le test structurel est la stratégie de choix pour tester un système critique. En fonction de la criticité du système, différentes applications de test structurel sont exigées (analyse de couverture structurelle, génération des données de test, preuve de la post condition). Cet article s'intéresse au problème de combinaison des applications de test structurel, peu touché jusqu'à maintenant, qui peut faciliter l'automatisation du processus de test structurel et réduire de manière significative le nombre de données de test générées. Pour intégrer plusieurs applications de test structurel en une seule approche, nous modélisons le programme sous test et son graphe de flot et de contrôle (GFC) par un problème de satisfaction de contraintes (PSC). Nous utilisons une nouvelle classification des sommets du GFC, la dépendance de contrôle, les techniques statiques d'assignation unique (SSA) et l'inférence de la programmation par contraintes. Le modèle PSC que nous proposons conserve toute la sémantique structurelle du programme, cette caractéristique le rendant utilisable pour différentes applications de test structurel : analyser une couverture structurelle, générer des données de test ou prouver la post condition. Nos expérimentations sur des benchmarks traditionnels montrent un gain de temps par rapport aux approches existanes de test structurel.

1 Introduction

Le test du logiciel est une étape importante dans le cycle de vie d'un logiciel. son coût peut facilement dépasser 50% du coût total d'un logiciel critique [16]. Les principales raisons qui obligent les fabricants de systèmes critiques à tester leurs produits sont les coûts énormes et les dommages qui peuvent être causés par un bogue logiciel : un comportement indésirable du système peut mener à une catastrophe. Un exemple célèbre est survenu en 1996, quand un bogue dans le composant de référence inertiel de la fusée a causé la destruction d'Ariane 5 juste 40s après le lancement. L'Agence spatiale européenne a annoncé un retard du projet, a estimé une perte économique directe de 500 millions de dollars US [1] et, implicitement, une perte de confiance des clients aux avantages de la concurrence. Ce genre de bogue est difficile à accepter parce qu'il conduit à des pertes financières énormes. Il pourrait également avoir des effets négatifs sur l'environnement ou mettre la vie ou la santé humaine en danger. Dans la catégorie des systèmes critiques, la sécurité humaine ou environnementale est largement basée sur les fonctionnalités du système. Afin d'augmenter la confiance en ces systèmes, différentes normes existantes (i.e., DO-178B pour l'aéronautique, la CEI 61513 pour le nucléaire, la CEI 50126 pour le ferroviaire). Une norme exige un niveau minimal de couverture de test pour une fonctionnalité du système en fonction de sa criticité. Pour la norme DO-178B [17], les programmes sont classés sur cinq niveaux (de E à A), chaque niveau représente l'effet possible d'une erreur du programme sur la sécurité des passagers. Le critère de couverture d'un programme est défini selon son classement dans la norme [14, 17] : un programme de niveau C doit être testé par un jeu de cas de test qui permette une couverture de toutes les instructions. Un programme de niveau B doit être testé par un jeu de données test qui permette une couverture de toutes les décisions. Un programme de niveau A doit être testé

par un jeu de cas de test qui permette une couverture de toutes les décisions conditions modifiées. Pour cette norme, tout cas de test doit être généré à partir des spécifications, ce qui veut dire que la principale tâche du test structurel est de mesurer la couverture des données de test, de montrer les parties du programme qui ne sont pas explorées et, dans certains cas, de générer le jeu de données de test complémentaire. Cette norme montre qu'un même programme au niveau A ou un système qui contient des programmes qui sont classés dans différents niveaux ont besoin d'une combinaison de plusieurs critères de couverture et plusieurs applications de test structurel (analyse ou génération de données de test).

Les principales applications de test structurel sont : l'analyse de la couverture structurelle, la génération des données de test pour satisfaire un critère de couverture ou atteindre un point donné dans un programme (i.e., générer une exception de division par zéro) et la preuve de post condition ou la génération d'un contre exemple. Généralement, le test d'un système critique exige une combinaison de ces applications et pour la certification, il est fortement recommandé de travailler sur un même outil de test [11]. Il est donc avantageux de combiner plusieurs applications de test structurel en une seule approche générique pour répondre aux diverses exigences de test structurel. Regrouper alors toutes les applications citées précédemment en une seule approche représente un défi très intéressant.

Dans cet article, nous proposons une nouvelle approche qui permet d'analyser la couverture structurelle, de générer des données de test structurel et de prouver la post condition ou de générer un contre exemple. Nous utilisons un problème de satisfaction de contraintes (PSC) qui modélise le programme sous test (PST) et son graphe de flot et de contrôle (GFC). L'approche est basée sur de nouveaux mécanismes qui utilisent la forme SSA [5, 8] et les dépendances de contrôle [8, 9]. L'utilisation de ces deux derniers mécanismes n'est pas nouvelle : ce qui est original dans ce travail est l'utilisation de la programmation par contraintes (PC) pour analyser une couverture structurelle, le regroupement des principales applications du test structurel en une seule approche et la modélisation du GFC par un PSC.

La suite de cet article est organisée comme suit : À la section 2, nous discutons les travaux connexes. À la section 3, nous introduisons les définitions et les concepts nécessaires pour l'approche. Les principes et les mécanismes utilisés pour concevoir un PSC sont expliqués à la section 4. La section 5 présente les résultats expérimentaux, en comparant notre approche à l'état de l'art et la section 6 présente des conclusions et des travaux futurs.

2 Travaux connexes

Dans la littérature, il existe des travaux qui sont liés à notre approche par l'utilisation de la PC pour répondre aux besoins d'un sous-ensemble d'applications du test structurel. La plupart de ces travaux sont orientés chemins [2, 18]. La technique orientée chemins consiste à extraire un chemin à partir de GFC ou d'une version instrumentée du PST, puis de générer un prédicat de ce chemin. La résolution de ce prédicat génère des données de test correspondant au chemin choisi. Cette procédure est répétée pour chaque élément (chemin, branche) de l'objectif de test (tous les chemins, toutes les branches). Nous signalons deux faiblesses dans cette technique : d'abord, le processus de génération de prédicat a un coût important lié à l'évaluation symbolique, en deuxième lieu, en général, cette technique ne peut pas générer des cas de test pour atteindre un point donné dans un PST. PathCrawler [18] et Osmose [2] sont deux outils qui peuvent générer un jeu de données de test pour une couverture de tous les chemins ou les k-chemins (PathCrawler pour les programmes en C, Osmose pour les exécutables). Ces approches peuvent être appliquées aux autres langages de programmation mais elles ne permettent pas d'analyser une couverture structurelle, de générer des données de test pour d'autres critères de couverture ou de prouver la post-condition.

Il existe aussi des approches orientées buts [4, 6, 12, 11] qui génèrent des données de test pour atteindre une instruction, une branche ou un chemin bien déterminé. En règle générale, ces approches traduisent un PST en un PSC en passant par un modèle SSA pour éviter le coût de l'évaluation symbolique. Contrairement aux approches orientées chemins, cette technique identifie difficillement des points du PST nécessaires pour atteindre un critère de couverture donné. Collavizza et al. [6] propose un cadre de travail pour prouver la post-condition d'un PST en Java ; une version améliorée a été implémentée dans [7], cette approche peut être utilisée pour prouver la satisfaction de la post-condition pour différents langages de programmation mais elle ne permet pas d'analyser une couverture structurelle ni de générer des données de test. INKA [4, 12] est un outil pionnier qui utilise la PC pour générer des données de test pour les programmes C. Les auteurs d'INKA proposent une approche qui peut être utilisée indépendamment du langage de programmation mais cette approche ne permet pas d'analyser une couverture structurelle ni de générer des données de test pour certains types de critères de couverture structurelle (i.e., la couverture tous les chemins, decision/condition modifiée). Euclide [11] est le seul outil basé sur la PC et orienté buts qui regroupe trois applications du test

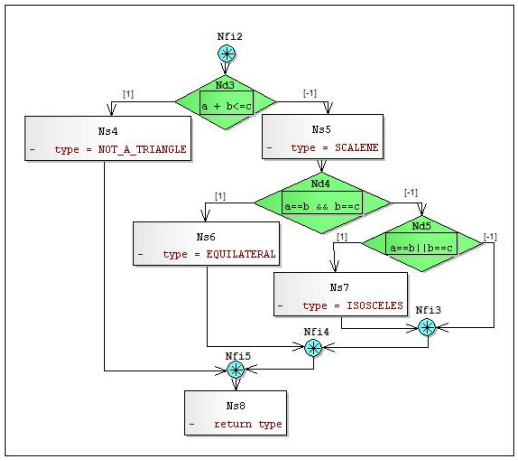


FIGURE 1 – Une partie du GFC de la fonction *tri_type* (lignes 13 à 29).

structurel. Il permet de prouver la post-condition et de générer les données de test pour quelques critères de couverture : toutes les instructions et toutes les décisions, mais avec une identification explicite des points à atteindre dans le code. Cet outil ne permet pas l'analyse d'une couverture structurelle et ne permet pas de générer des données de test pour d'autre mèsures de couverture tel que tous les k-chemins.

Notre approche combine les deux techniques, on peut dire qu'elle est orientée but et chemin. En utilisant la forme SSA, notre approche ne souffre pas du coût de l'évaluation symbolique ni de l'identification des points requis pour un critère de couverture donné, qui est résolu par la modélisation du GFC. En outre, elle est applicable pour répondre aux besoins variés du test structurel : analyser une couverture structurelle, générer de données de test pour atteindre un point dans le PST ou satisfaire un critère de couverture et de prouver la post-condition ou générer un contre-exemple.

3 Notions de base

Le programme triangle, utilisé dans des travaux antérieurs [2, 4, 15, 18], permet à l'utilisateur d'identifier un triangle par la longueur de ses trois côtés. Le code du Prog.1 est une implémentation possible de ce programme [15]. Cette implémentation est différente de celle utilisée dans [4, 6, 18]. Nous utilisons le programme triangle pour exprimer, définir et illustrer notre approche.

Definition 1. *Dans un graphe de flot et de contrôle (GFC) [9], nous distinguons cinq classes de noeuds : noeud de décision, noeud de jointure, noeud d'instruction et noeud des paramètres et variables globales.* Un noeud de décision *est un noeud qui a deux ou plusieurs arêtes sortantes. Il représente une instruction de contrôle. Sur la Fig. 1* $Nd3$, $Nd4$ *et* $Nd5$ *sont des noeuds de décision.* Un noeud de jointure *est un noeud qui a deux ou plusieurs arêtes entrantes. Il représente la fin d'une instruction de contrôle. Sur la Fig. 1* $Nfi2$, $Nfi3$, $Nfi4$ *et* $Nfi5$ *sont des noeuds de jointure.* Un noeud de condition *est un noeud dérivé d'un noeud de décision. Si une expression logique d'un noeud de décision est composée de plusieurs conditions, alors la décomposition de cette expression à des conditions atomiques génère un ensemble de ces noeuds. Sur la Fig. 1, cette classe de noeuds est cachée dans les décisions* $Nd4$ *et* $Nd5$. Un noeud d'instruction *est un noeud qui accepte une seule arête entrante et une autre sortante. Il représente une séquence d'instructions qui ne contient aucune instruction de contrôle. Sur la Fig. 1,* $Ns4$, $Ns5$, $Ns6$, $Ns7$ *et* $Ns8$ *sont des noeuds d'instruction.* Une décision positive *(resp.* une décision négative*) est une branche qui satisfait (resp. ne satisfait pas) la condition d'un noeud de décision. Dans notre exemple, la Fig.1, l' arête de* $Nd3$ *à* $Ns4$ *est une décision positive et celle de* $Nd3$ *à* $Ns5$ *est une décision négative. Nous utilisons le signe* "$+$" *pour désigner la décision positive et le signe* "$-$" *pour la décision négative.* Un noeud actif *est un noeud*

Prog. 1 le code source de la fonction *tri_type* en language C.

```
01 int tri_type(int a, int b, int c)  02 {
03    int type;
04    int t;
05    if (a > b){
06         t = a; a = b; b = t;       07 }
08    if (a > c){
09         t = a; a = c; c = t;       10 }
11    if (b > c){
12         t = b; b = c; c = t;       13 }
14    if (a + b <= c){
15          type = 0;     16 }
17    else 18 {
19         type = 1;
20         if (a == b && b == c){
21              type = 2; 22 }
23         else {
24              if (a == b || b == c){
25                   type = 3; 26 }
27         } 28 }
29    return type;
30 }
```

qui fait partie du chemin d'exécution sélectionné, tout noeud qui n'appartient pas au chemin sélectionné est appelé noeud neutre.

Definition 2. Une dépendance de contrôle *[8] est une relation entre deux noeuds $N1$ et $N2$. $N2$ est en dépendance de contrôle avec $N1$ si et seulement si l'exécution de $N2$ dépend du résultat de l'exécution du noeud $N1$. Par exemple, sur la Fig. 1, $Ns4$ est en dépendance de contrôle avec $Nd3$.*

Definition 3. Une contrainte gardée *[13], également appelée contrainte conditionnelle, est une contrainte composée de deux parties : une garde et un objectif. L'ajout de l'objectif au PSC dépend de l'état de la garde : si la garde est impliquée (satisfaite) par l'état du PSC, alors l'objectif est inséré dans le PSC et la contrainte gardée est enlevée ; si la négation de la garde est impliquée par l'état du PSC, alors la contrainte gardée est retirée de PSC ; si le solveur ne peut pas prouver l'une ou l'autre, il met la contrainte gardée en état de suspension.*

Definition 4. La forme SSA *[5, 8] est une représentation formelle d'un programme, elle conserve la sémantique du programme. Elle définit les variables de façon unique à chaque instruction d'affectation et chaque utilisation peut être atteinte depuis cette définition. Par exemple, pour une variable V, une nouvelle variable dérivée V_i est générée à chaque nouvelle définition de V rencontrée dans le programme. La variable V est remplacée par V_i dans toute instruction qui utilise V jusqu'à la définition suivante de V. Ce processus de renommage est géré par une fonction σ : pour simplifier, $\sigma(V)$ renvoie le nombre d'instructions d'affectation de V qui précédent l'instruction appelante. Dans un programme qui contient plusieurs branches, les variables dérivées d'une même variable peuvent atteindre le même point de jointure. Ces variables dérivées doivent être fusionnées en une seule variable qui est utilisée à partir de la jointure. Cette variable est générée par une fonction φ : si une instruction de contrôle atteint son noeud de joiture via sa branche numéro i, la valeur retournée par cette fonction est celle de l'opérande numéro i. Une fonction φ est ajoutée à la fin de chaque instruction de contrôle.*

Mod. 1 Modèle SSA de la fonction *tri_type*.

```
Np0 int tri_type(int a0, int b0, int c0){
Ns0     int type0=0;
Ns0     int t0=0 ;
Nd0     if (a0 > b0){
Ns1           t1 = a0; a1 = b0; b1 = t1; }
Nfi0    t2=fi(t1,t0); a2=fi(a1,a0);
Nfi0            b2=fi(b1,b0);
Nd1     if (a2 > c0){
Ns2           t3 = a2; a3 = c0; c1 = t3; }
Nfi1    t4=fi(t3,t2); a4=fi(a3,a2);
Nfi1            c2=fi(c1,c0);
Nd2     if (b2 > c2){
Ns3           t5 = b2; b3 = c2; c3 = t5; }
Nfi2    t6=fi(t5,t4); b4=fi(b3,b2);
Nfi2            c4=fi(b3,b2);
Nd3     if (a4 + b4 <= c4)
Ns4          type1 = 0;
        else{
Ns5          type2 = 1;
Nd4          if (a4 == b4 && b4 == c4)
Ns6               type3 = 2;
             else{
Nd5               if (a4 == b4 || b4 == c4)
Ns7                    type4 = 3;
Nfi5              type5=fi(type4, type2);
             }
Nfi4       type6=fi(type3,type5);
        }
Nfi3    type7=fi(type1,type6);
Ns8     return type7;
    }
```

un noeud et une instruction. Les noeuds du GFC sont représentés par des variables entières avec un petit domaine $\{-1, 0, 1\}$: -1 signifie que le noeud est actif et sa décision est négative, 0 signifie que le noeud est neutre ; 1 signifie que le noeud est actif et sa décision est positive. Pour répondre à un besoin de test structurel d'un PST en utilisant notre approche, nous proposons quatre étapes principales : le modèle SSA ; la modélisation par contraintes du GFC ; la construction de PSC global ; et la résolution du PSC.

4 Approche

Les classes des noeuds et les relations de dépendance sont la base de notre approche pour modéliser un PST par un PSC. Un PST est traduit en un ensemble de contraintes dont la majorité sont des contraintes gardées. Ces contraintes sont obtenues par la traduction d'une instance de relation entre deux noeuds ou entre

4.1 Le modèle SSA

Si un PST contient deux instructions d'assignation d'une même variable à deux valeurs différentes (i.e., $x = 1$ et $x = 2$), la traduction directe de ces deux instructions en contraintes génère deux contraintes incohérentes. Pour éviter ce problème bien connu

[6, 12, 11], avant de traduire le PST en un PSC, on le traduit en un modèle SSA. Le Mod. 1 montre le modèle SSA de la fonction de *tri_type*.

4.2 La modélisation par contraintes du GFC

Cette étape consiste à modéliser le GFC du PST par un PSC préliminaire. Nous commençons par la génération d'un GFC qui est caractérisé par des noeuds indépendants pour chaque classe. Puis, nous identifions les noeuds du GFC selon leur classe d'appartenance et leur ordre dans cette classe : un noeud de décision est identifié par un préfixe Nd et un indice i ; un noeud de condition est identifié par un préfixe correspondant au noeud de décision d'origine Ndi et un indice j ; un noeud d'instructions est identifié par un préfixe Ns et un indice i. Sur le Mod. 1 à la gauche de chaque instruction, nous mentionnons le noeud correspondant du GFC. Après l'identification des noeuds, nous étiquetons les arrêtes selon leur noeud d'origine : une arrête sortante d'un noeud d'instruction est étiquetée par 1 ; une arête sortante d'un noeud de condition ou de décision est étiquetée par 1 si la décision est positive ou par -1 si la décision est négative.

Table 1 – Noeud-Noeud : Table de contraintes

Nd_i	N_j **dans** Nd_i-	N_k **dans** Nd_i+
0	0	0
1	0	-1
1	0	1
-1	-1	0
-1	1	0

Table 2 – Décision conjonctive : Table de contraintes

Nd_i	Nd_{i1}	...	Nd_{ij}	...	Nd_{in}
1	1	1	1	1	1
0	0	0	0	0	0
-1	-1	*	*	*	*
...	...	...	...	...	...
-1	*	*	-1	*	*
...	...	...	...	...	...
-1	*	*	*	*	-1

$Noeud - Noeud$ est une relation exprimée par deux noeuds, dans laquelle au moins l'un des deux est un noeud de décision. Lorsqu'un noeud est dans l'une des branches d'un noeud de décision, le premier ne peut être neutre si cette branche est active. Cette relation est alors exprimée par les contraintes illustrées dans la Tab. 1. La relation $Noeud - Noeud$ est la règle principale pour obtenir le modèle à contraintes d'un GFC.

Afin de montrer les branches cachées d'un noeud de décision multi-conditions, à partir de ce noeud de décision, nous dérivons un graphe de décisions en décomposant l'expression multi-conditions en conditions atomiques. Une décision multi-conditions utilise deux formes d'expressions logiques : la forme conjonctive ou la forme disjonctive. Chaque forme peut être exprimée en termes de ses conditions atomiques : Une forme conjonctive (resp. disjonctive) est vraie (resp. fausse) si et seulement si toutes ses conditions atomiques sont vraies (resp. fausses). La relation entre un noeud de décision conjonctif Nd_i et ses noeuds de conditions dérivés Nd_{ij} est exprimée par les contraintes dans la Tab. 2. La relation entre un noeud décision disjonctif Nd_i et ses noeuds de conditions dérivés Nd_{ij} est exprimée de façon similaire à la forme conjonctive. Pour obtenir le tableau des contraintes d'une décomposition disjonctive, il suffit de permuter les 1 et les -1 dans la Tab. 2.

Pour les expressions complexes qui combinent des formes conjonctives et disjonctives, il est préférable d'utiliser des noeuds intermédiaires. La décomposition des noeuds de décision n'est pas nécessaire si le besoin de test n'exige pas le niveau de condition.

Lors de la génération de PSC préliminaire, un noeud est traduit par une variable dont le domaine est l'ensemble d'étiquettes de ses arêtes sortantes, sauf pour les noeuds de jointure qui prennent le domaine du noeud de décision correspondant. Si un noeud n'est pas la racine, on ajoute la valeur 0 à son domaine. Nous utilisons les règles de décomposition et la règle $Noeud - Noeud$ pour générer des contraintes qui expriment les relations entre les noeuds. Le modèle Mod.2 montre le PSC résultat de la modélisation du GFC de la fonction *tri_type* : la première partie (01-04) contient les définitions des variables et de leurs domaines ; la deuxième partie (05-12) exprime les relations entre noeuds du GFC, elle a été générée selon la règle $Noeud - Noeud$; la troisième partie (13-16) exprime la relation entre les décisions et leurs noeuds de conditions dérivés. Elle a été générée en utilisant les règles de décomposition.

4.3 Construction du PSC global

La troisième étape utilise le PSC préliminaire, le modèle SSA et la relation entre un noeud et les instructions qu'il représente pour créer un nouveau PSC global. Elle consiste à traduire chaque instruction de modèle SSA en une ou plusieurs contraintes. Dans cette section, nous discutons aussi certaines particularités des boucles et des expressions d'affectation et de déclaration des variables.

$Noeud - Instruction$ est une relation exprimée sur un noeud d'instruction et son contenu. Une instruction peut être considérée comme une contrainte qui est nécessairement satisfaite si son noeud est actif. Il

Mod. 2 PSC du GFC de la fonction *tri_type*.

```
01 X{Nsm , Ndj, Nfij, Nd4k, Nd5k
          / 0<=m<=8, 0<=j<=5, 0<=k<=2};
02 D(Ns0)=D(Ns8)={1}; D(Nsi / 0<i<=7)={0,1};
03 D(Ndj)=D(Nfij)={-1,1} / 0<=j<=3;
04 D(Ndj)=D(Nfij)=D(Ndjk)= {-1,0,1}
          / 2<=j<=5, 0<=k<=1;
05 C{ //les contraintes suivantes sont
     //générées selon la règle Noeud-Noeud
06      table (Nd0,Ns1, Tab.1);
07      table (Nd1,Ns2, Tab.1);
08      table (Nd2,Ns3, Tab.1);
09      table (Nd3,Nd4, Tab.1);
10      table (Nd4,Ns6, Tab.1);
11      table (Nd5,Ns7, Tab.1);
12      Nd0=Nfi0; Nd1=Nfi1; Nd2=Nfi2;
       Nd3=Nfi3; Nd4=Nfi4; Nd5=Nfi5;
13 //les contraintes suivantes sont générées
    // selon la règle de décomposition
14      table (Nd4, Nd40, Nd41, Tab.2);
15      table (Nd5, Nd50, Nd51, Tab.2);
16 }
```

existe une relation entre un noeud d'instruction et son contenu : les instructions qu'il représente. Si un noeud Ns_i est actif (prend la valeur 1), toute instruction S de ce noeud est exécutée. $Noeud-Condition$ est une relation exprimée sur un noeud de décision et son instruction de contrôle : si le noeud est actif, alors sa condition prend une valeur vrai ou faux selon la décision prise. Il existe une relation d'implication entre un noeud de décision Nd et la condition de son instruction de contrôle C. Pour exprimer les relations entre un noeud de GFC et son contenu d'instructions, nous utilisons les règles $Noeud - Instruction$ et $Noeud - Condition$. Chaque instruction est traduite en au moins une contrainte : une définition d'une variable dans le noeud paramètre se traduit par la définition de la variable correspondante dans le PSC. Dans un noeud d'instruction Ns_i, une instruction d'affectation S (i.e., $type_2 = 1$) se traduit par une contrainte gardée dont la garde est une expression logique d'égalité entre la variable Ns_i et la valeur 1 (i.e., $Ns_5 = 1 \rightarrow type_2 = 1$). Si un noeud d'instruction est neutre ($Ns_i = 0$), une variable qui est définie dans ce noeud peut forcer le solveur à générer un nombre important de solutions symétriques. Pour éviter ce problème, nous affecttons à cette variable sa dernière valeur avant l'instruction de contrôle parent(i.e., $Ns_5 = 0 \Rightarrow type_2 = type_0$). Dans un noeud de décision ou de condition Nd_i, la condition de son instruction de contrôle C (i.e., $a_4 + b_4 < c_4$) se traduit par deux contraintes gardées, la première (resp. la seconde) utilise C (resp. négation de C) comme partie objectif et l'expression correspond à la décision positive (resp. négative) du noeud en tant que garde (i.e., $Nd_3 = 1 \Rightarrow b_4 + a_4 < c_4; Nd_3 = -1 \Rightarrow not(a_4 + b_4 < c_4)$). Dans un noeud de jointure, une fonction φ se traduit par une contrainte gardée pour chaque valeur dans son domaine (i.e., $Nfi_4 = 1 \Rightarrow type_6 = type_3; Nfi_4 = -1 \Rightarrow type_6 = type_5; Nfi_4 = 0 \Rightarrow type_6 = type_2$).

Déclaration des variables

Une déclaration de variable V_0 de type $T(entier, caractere, booleen)$ se traduit dans le PSC global en une nouvelle variable V_0 de type T. Le domaine de V_0 est extrait par l'analyse du PST, il peut être borné entre deux valeurs V_{min} et V_{max}, sinon le domaine contient toutes les valeurs de T supportées par le système (i.e., sur un système 32-bit, de type entier est définie dans l'intervalle $[-2^{31}, 2^{31} - 1]$).

Un tableau de taille fixe est traduit dans le PSC par un tableau de même taille. Une déclaration d'un tableau t_0 de taille variable (allocation dynamique) se traduit dans PSC par un tableau de taille prédéfinie l_{max} et une variable l_{t0} qui représente sa taille réelle. Pour ignorer les indices qui sont supérieurs à l_{t0}, nous ajoutons quelques contraintes : ces variables sont fixées à une constante et sont ignorées par toute autre contrainte sur le tableau.

Instructions et expressions d'affectation

Nous distinguons deux types d'instructions d'affectation : celles qui sont exprimés sur des variables scalaires, et celles qui sont exprimées sur des variables de type tableau.

1. Sur une variable scalaire : l'instruction se traduit par une déclaration d'une nouvelle variable et une contrainte gardée conformément à la règle $Noeud - Instruction$.

2. Sur une variable tableau : dans un programme, un tableau est manipulé par deux groupes d'instructions : lecture et écriture. La lecture d'un élément ne pose aucun problème, car l'instruction est traduite de la même manière qu'une instruction simple, c'est-à-dire que la variable du tableau est remplacée par la variable équivalente. Dans le cas d'une écriture d'un élément, une nouvelle variable du tableau est générée qui contient les mêmes éléments que son prédécesseur sauf l'élément qui contient une nouvelle donnée. Par exemple, l'af-

fectation $t[i] = x$ est traduite selon la formule :

$$T_{\sigma(T)}[k] = \begin{cases} x_{\sigma(x)} & \text{si } k = i \\ T_{\sigma(T)-1}[k] & \text{sinon} \end{cases}$$

Boucles

Toute boucle du PST doit être transformée en une boucle $tant - que$ ($While$). Nous utilisons une constante k ($k - chemin$) afin de limiter le nombre d'itérations dans une boucle. Avec cette limitation, une boucle contient $k + 1$ noeuds de décision. Pour forcer le PST à quitter la boucle au maximum après k itérations, le dernier noeud de décision Nd_{k+1} doit toujours être différent de la valeur 1. Le résultat de la traduction d'une boucle ($While\ C\ B$;) est :

1. $Nd_0 \neq 0$;
2. $\forall\ 0 < i \leq k+1,\ Nd_i \neq 0 \Leftrightarrow Nd_{i-1} = 1$;
3. $\forall\ 0 \leq i \leq k+1$
   - $Nd_i = 1 \Leftrightarrow Ns_i = 1$, où Ns_i est le noeud d'instruction qui contient B à la i^{eme} itération ;
   - $Nd_i = Nfi_i$, où Nfi_i est le noeud de jointure correspond au noeud de décision Nd_i ;
   - $Nd_i = 1 \Rightarrow C_{\sigma i(C)}$, où $C_{\sigma i(C)}$ est la forme SSA de C à la i^{eme} itération ;
   - $Nd_i = -1 \Rightarrow not(C_{\sigma i(C)})$;
4. $Nd_{k+1} \neq 1$.

4.4 Résolution du PSC

La méthode de traduction proposée donne un PSC qui maintient toute la sémantique structurelle du PST. Cette propriété rend notre approche capable de répondre aux différents besoins de test structurel : analyser ou générer tout type de couverture de test structurel et prouver la post-condition ou générer un contre-exemple. Une restriction sur le PSC global par quelques contraintes supplémentaires ou par une stratégie de recherche permet de limiter et d'orienter le PSC vers un ensemble spécifique de solutions pour répondre à des besoins variés de test.

Ensemble objectif de test (EOT)

En règle générale, l'objectif d'un test structurel consiste à analyser ou exécuter un ensemble de chemins qui répond à un critère de couverture simple ou combiné. Cet objectif peut être décomposé en un ensemble d'objectifs partiels exprimables en termes de noeuds de décision, noeuds de condition, ou noeuds d'instruction. Ainsi, un objectif partiel est une conjonction de conditions sur ces trois types de variables. Alors, nous pouvons représenter un objectif partiel par un ensemble de paires <$variable$,$valeur$> (i.e., l'objectif est l'exécution de Ns_2 exprimée par $\{<Ns_2,1>\}$). Un objectif partiel peut contenir des paires de la même classe ou des classes combinées. Dans notre approche, l'objectif de test est représenté par un EOT qui contient des objectifs partiels. Nous donnons les EOT pour les mesures de couverture les plus utilisées. Le recensement ci-après n'est pas exhaustif.

1. Couverture de toutes les instructions : chaque objectif partiel est un ensemble qui contient une seule paire composée d'une variable noeud d'instruction et la valeur 1. L'EOT contient tous les objectifs partiels possibles (i.e., $\{\{<Ns_i, 1>\}/\ 0 \leq i \leq 8\}$).
2. Couverture de toutes les décisions : chaque objectif partiel est un ensemble singleton qui contient une paire composée d'une variable noeud de décision et la valeur 1 ou -1. L'EOT contient tous les objectifs partiels possibles (i.e., $\{\{<Nd_i, 1>\}, \{<Nd_i, -1>\}/\ 0 \leq i \leq 5\}$).
3. Nous utilisons la même méthode pour générer l'EOT pour d'autres objectifs de test ou mesures de couverture : toutes conditions, conditions/decisions, multi-conditions/décisions. Nous utilison l'algorithme proposé dans [10] pour générer l'EOT de la couverture conditions/décisions modifiées.

Génération des données de test

Pour générer des données de test en fonction d'un objectif, nous devons d'abord construire l'EOT. Tant que cet ensemble n'est pas vide, on choisit un objectif partiel, on l'enlève et on fixe ses variables à leurs valeurs, puis nous cherchons une solution. Si le PSC n'a pas de solution, nous marquons cet objectif partiel comme irréalisable. En général, la preuve d'un PSC irréalisable est un problème indécidable, alors durant la recherche, nous montrons partiellement qu'il est irréalisable. Si le problème a une solution, nous vérifions si d'autres objectifs partiels sont satisfaits par cette solution, dans le cas échéant, nous supprimons ces objectifs partiels de l'EOT.

Analyse structurelle

Un jeu de données à analyser est représenté par un ensemble de données de test (EDT) qui contient des vecteurs de test $< val_1, ..., val_n >$, où val_i est la valeur du paramètre d'entrée numéro i. Pour mesurer ou analyser la couverture structurelle d'un jeu de données de test, nous devons d'abord construire l'EOT correspondant à l'objectif de test. Tant que l'EDT et l'EOT ne sont pas vides, on choisit un vecteur de données et on l'enlève de l'EDT, dans le PSC nous fixons les variables paramètres à leurs valeurs, puis nous cher-

TABLE 3 – Couverture toutes les décision : comparaison avec l'approche Gotlieb, et al. sur le programme *try_type*.

Decision		**Nd0**		**Nd1**		**Nd2**		**Nd3**		**Nd4**		**Nd5**	
Value		1	-1	1	-1	1	-1	1	-1	1	-1	1	-1
Time (s)	$CSP1$	0.01	0.01	0.77	0.01	0.01	0.01	0.01	>300	>300	>300	>300	>300
	$CSP2$	0.01	0.75	0.95	0.75	0.09	0.75	0.01	0.75	0.01	0.75	0.01	0.75

chons une solution. Si le problème n'a pas de solution, nous marquons ce vecteur hors domaine. Si le problème a une solution, nous vérifions s'il existe des objectifs partiels qui sont satisfaits par cette solution ; le cas échéant, nous supprimons ces objectifs partiels de l'EOT. Enfin, si l'EOT devient vide, alors une couverture de 100% est atteinte, sinon nous calculons le pourcentage de couverture. Pour le sous-ensemble de l'EOT qui n'est pas couvert, nous générons des données de test et les chemins qui ne sont pas couverts.

Prouver une post-condition

Nous formulons la contrainte équivalente de la post-condition à prouver en remplaçant les variables de PST par les variables équivalentes du PSC, puis nous insérons la négation de cette contrainte dans ce PSC [6]. Si le problème a une solution, alors il existe un chemin qui viole cette post-condition. La solution donne le chemin concerné et les données de test pour violer la post-condition. Si le problème n'a pas de solution, alors le post-condition est toujours satisfaite.

Génération des données de test pour une couverture k-chemins

Dans notre approche, un chemin d'exécution est une affectation des variables noeud de décision. Un premier chemin peut être couvert par la première solution atteinte par le solveur pour le CSP global. À partir de cette solution, nous pouvons construire le prédicat de ce chemin qui est une conjonction des expressions d'égalité entre les variables noeud de décision et leurs valeurs. L'insertion de la négation de ce prédicat dans le PSC global force le solveur à donner une autre solution correspondant à un chemin différent. Nous utilisons ce mécanisme : pour chaque solution trouvée, la négation de son prédicat est insérée dans PSC global, jusqu'à ce qu'il devienne irréalisable, ce qui signifie que tous les chemins faisables sont couverts.

Contrairement au principe basé sur l'EOT, chaque chemin (objectifs partiel) est un PSC à résoudre, ce qui signifie un PSC irréalisable pour chaque chemin infaisable, qui est également adoptée par plusieurs autres approches [6, 7, 18]. Notre approche n'a qu'un seul PSC irréalisable à résoudre.

Pour réaliser ce mécanisme, nous définissons une stratégie de recherche : dans un premier niveau de recherche l'énumération est faite sur les variables noeud de décision (Nd_i), dans un deuxième niveau de recherche l'énumération est faite sur les variables qui représentent les paramètres d'entrée. Une fois qu'une solution est trouvée, nous obligeons le solver à faire un retour-arrière vers le premier niveau.

5 Comparaisons avec l'état de l'art

Nos comparaisons sont faites en fonction du temps nécessaire pour générer un jeu de données de test pour un critère de couverture spécifique. Toutes les expériences ont été réalisées avec ILOG OPL Studio 3.7.1, sur un Windows XP, Intel Core 2 Duo 2 GHz, 2 Go de mémoire. Pour la première comparaison, nous avons limité le temps de recherche à 5 minutes.

Couverture toutes décisions

Pour comparer notre approche avec celle de Gotlieb et al. [12], nous avons traduit manuellement l'implémentation du programme triangle proposée par [15] qui est différente de celle utilisée par [12, 18] en deux PSC. Dans une première version, $PSC1$, nous avons utilisé l'approche proposée par Gotlieb et al. et dans une seconde version, $PSC2$, nous avons utilisé notre approche. La fonction *tri_type* contient six instructions de contrôle. Notre objectif est de satisfaire une couverture de toutes les décisions. Pour chaque PSC, nous avons créé un ensemble de douze contraintes équivalentes, une par branche, et avons ajouté à chaque version une contrainte à la fois. La Tab. 3 fournit les détails des résultats obtenus.

Pour les sept premières branches, le temps nécessaire pour résoudre le $PSC1$ est légèrement meilleur que le temps nécessaire pour résoudre notre $PSC2$, l'une des raisons étant que l'approche de Gotlieb et al. ne contient que des variables de PST, alors que notre approche contient également des variables de noeud, ce qui nécessite un délai supplémentaire pendant la recherche. Une autre raison est que ces sept branches sont faciles à couvrir (non-triangle, scalènes). Toutefois, notre approche est plus efficace dans les cinq dernières branches, qui sont plus compliquées à satisfaire (équilatéral, isocèle). Sur les douze branches, après cinq minutes d'attente pour chaque branche, $PSC1$ n'a pas pu générer des données de test pour couvrir les cinq branches qui représentent 40 % du PST, tan-

TABLE 4 – Couverture de tous les chemins : comparaison avec PathCrawler [18].

Program	k-chemins	#Ch. faisables	Notre Approche		PathCrawler [18]	
			#D. de test	temps (s)	#D. de test	Temps (s)
try_type	-	10	10	0.001	14	0.010
Merge	2	17	17	0.080	19	-
Merge	5	321	321	0.148	337	0.780
Merge	10	20481	20481	28.640	20993	116.000
Sample	3	240	240	0.060	241	0.270

TABLE 5 – Couverture de tous les chemins : comparaison avec PathCrawler [3] sur le programme *Merge*.

k-chemins	#Ch. faisables	Notre approche		PathCrawler [3]	
		#D. de test	Temps (s)	#D. de test	Temps (s)
2	17	17	0.080	19	0.330
5	321	321	0.148	337	0.800
10	12287	12287	18.163	12798	37.200
15	204931	204931	827.250	216371	876.350
All-Paths(19)	705431	705431	5486,953	705431	3407,980

dis que notre approche a généré des données de test pour chaque branche.

Couverture k-chemins

Nous avons utilisé notre approche de couverture de tous les k-chemins pour traduire manuellement les programmes présentes dans la Tab. 4, puis nous avons comparé nos résultats avec ceux rapportés dans [18] (Tab. 4) et [3] (Tab. 5).

Par définition, un vecteur de données de test qui est généré pour une couverture de k-chemins ne doit pas dépasser k itérations dans une boucle. Le mécanisme de génération de prédicat d'un chemin selon son identification partielle (chemin incomplet) utilisé par PathCrawler peut générer des données de test qui dépassent cette limite. Ces données de test superflues expliquent la différence du nombre de données de test par rapport à notre approche. Nous avons généré le nombre exact de données de test équivalent au nombre de chemins réalisables, alors que PathCrawler a généré des données de test supplémentaires (512 données de test superflues pour $k = 10$), mais ces données ne sont pas compatibles avec la valeur de k choisie.

En termes de temps d'exécution, notre approche a prouvé son efficacité en particulier pour les deux derniers programmes (*Merge* et *Sample*) où il est presque cinq fois plus rapide que PathCrawler, ce qui veut dire que même si nous supposons que notre machine est 5 fois plus rapide que celle utilisée pour les expérimentations de PathCrawler, ce qui loin d'être le cas, nos résultats restent comparables. Vu que nous avons traduit manuellement les programmes, nous avons ignoré le temps nécessaire pour cette tâche parce qu'il n'est pas significatif par rapport au temps nécessaire pour la résolution du PSC global.

La génération des données de test pour couvrir tous les chemins sans limiter le nombre d'itérations présentée dans Tab. 5 montre que PathCrawler est légèrement plus efficace. Pour une valeur de k supérieure à dix, le nombre d'itérations des deux dernières boucles du programme *Merge* ne dépasse pas dix. Mais notre approche utilise une seule valeur pour k, ce qui signifie un nombre supplémentaire de chemins infaisables à prouver. Pour un k inférieur ou égal à dix, notre approche est nettement meilleure que PathCrawler [3]. En termes pratiques, pour un PST qui contient des boucles, le test structurel limite le nombre d'itérations à un petit nombre qui est généralement égal à deux ou trois. Ainsi un critère de couverture tous-chemins sans limitation sur le nombre d'itérations n'est pas réaliste [16]. Nous observons également que, pour $k = 15$, PathCrawler a généré un ensemble de 11.440 cas de test supplémentaires qui doivent alors être appliqués inutilement.

6 Conclusion

Dans cet article, nous avons présenté une nouvelle approche pour différentes applications de test structurel. Il s'agit d'une approche qui combine les applications de test structurel les plus utilisées. Sa nouveauté réside dans l'utilisation de la PC pour l'analyse structurelle, la combinaison d'un nombre important d'applications de test structurel, la combinaison de tous les critères de couverture structurelle qui sont basés sur le flot de contrôle et la modélisation par contraintes

d'un GFC. Une nouvelle restructuration du GFC et une classification de ses noeuds ont été données. Nous avons montré que la modélisation d'un PST combiné à son GFC par un PSC conserve sa sémantique structurelle et peut donc répondre aux différents besoins de test structurel : analyse structurelle, génération des données de test et preuve de post-condition. Nous avons déjà donné, à la section 4.4, certaines façons dont notre approche peut être utilisée pour répondre aux besoins variés de test structurel. Notre approche peut être utilisée pour automatiser le processus de test structurel, réduire la taille d'un ensemble de données de test et réduire ainsi le temps nécessaire pour tester un système critique.

Les résultats obtenus en comparant notre approche à des approches de la littérature sur différents benchmarks sont très prometteurs, ce qui peut être considéré comme un très bon point de départ pour une automatisation complète du processus de test structurel sur la base de PC. L'efficacité de notre approche est fortement dépendante des performances du solveur, ces dernières sont limitées par le nombre de variables et la taille de leurs domaines. Dans le futur, nous allons nous concentrer à la fois sur l'implantation d'un outil complet qui mettra en oeuvre cette approche et sur son extension pour dépasser ses limites actuelles. En particulier, nous aimerions traiter les pointeurs, les nombres à virgule flottante et les appels aux fonctions, qui sont une extension difficile mais importante pour traiter des systèmes industriels.

Références

[1] Douglas N. Arnold. The explosion of the ariane 5. `http://www.ima.umn.edu/~arnold/disasters/ariane.html`, 2000. [Online ; accessed 09-Mai-2011].

[2] S. Bardin and P. Herrmann. Structural testing of executables. In *International Conference on Software Testing, Verification and Validation*, pages 22–31, 2008.

[3] B. Botella, M. Delahaye, N. Kosmatov, M. Roger P. Mouy, and N. Williams. Automating structural testing of c programs : Experience with pathcrawler. In *Fourth International Workshop on the Automation of Software Test*, pages 70–78.

[4] B. Botella, A. Gotlieb, C. Michel, M. Rueher, and P. Taillibert. Utilisation des contraintes pour la génération automatique de cas de test structurels. *Technique et sciences informatiques*, 21 :21–45, 2002.

[5] M. Brandis and H. Mössenböck. Single-pass generation of static single-assignment form for structured languages. *ACM Transactions on Programming Languages and Systems*, 14 :1684–1698, 1994.

[6] H. Collavizza and M. Rueher. Exploration of the capabilities of constraint programming for software verification. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 3920, pages 0302–9743, 2006.

[7] H. Collavizza, M. Rueher, and P. Van Hentenryck. Cpbpv : a constraint-programming framework for bounded program verification. 15 :238–264, 2010.

[8] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13 :451–490, 1991.

[9] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *Transactions on Programming Languages and Systems*, 9 :319–349, 1987.

[10] K. A. Foster. Sensitive test data for logic expressions. *ACM singsoft software engineering notes*, 9 :120–125, 1984.

[11] A. Gotlieb. Euclide : A constraint-based testing framework for critical c programs. In *International Conference on Software Testing, Validation and Verification*, pages 151–160, 2009.

[12] A. Gotlieb, B. Botella, and M. Rueher. A clp framework for computing structural test data. *Computational Logic*, 1861 :399–413, 2000.

[13] J. Jaffar and M. J. Maher. Constraint logic programming - a survey. *Logic Programming*, 20 :503–581, 1994.

[14] J. Kong and H. Yan. Comparisons and analyses between rtca do-178b and gjb5000a, and integration of software process control. In *International Conference on Advanced Computer Theory and Engineering*, volume 6, pages 367–372, 2010.

[15] P. McMinn. Search-based software test data generation : a survey. *Software Testing Verification & Reliability*, 14 :105–156, 2004.

[16] G. J. Myers. *The art of software testing.* John Wiley and Sons, 1979.

[17] RTCA. Software consideration in airborne systems and equipment certification. *Software verification process*, 1992.

[18] N. Williams, B. Marre, P. Mouy, and M. Roger. Pathcrawler : Automatic generation of path tests by combining static and dynamic analysis. In *European Dependable Computing Conference*, volume 3463, pages 281–292, 2005.

Régions intérieures et linéarisations par intervalles en optimisation globale

Gilles Trombettoni, Ignacio Araya, Bertrand Neveu, Gilles Chabert

INRIA, I3S, Université Nice–Sophia (France), UTFSM (Chili),
Imagine LIGM Université Paris–Est (France), LINA, EMN (France)

Gilles.Trombettoni@inria.fr, iaraya@inf.utfsm.cl, Bertrand.Neveu@enpc.fr, Gilles.Chabert@emn.fr

Abstract

Les communautés d'analyse par intervalles et de programmation (logique) par contraintes ont exploité les intervalles pour leur capacité à représenter des ensembles infinis de solutions dans les systèmes de contraintes continus. En particulier, les boîtes ou régions *intérieures* permettent de représenter des sous-ensembles de l'espace de recherche dans lesquels *tout* point est solution. Notre première contribution est l'utilisation d'algorithmes récents et nouveaux d'extraction de régions intérieures dans la phase d'amélioration du majorant (faisable) en optimisation globale sous contraintes.

La *relaxation linéaire* est également un ingrédient majeur, utilisé notamment pour minorer la fonction objectif. Nous avons adapté la *taylorisation sur intervalles convexe* – relaxation linéaire proposée par Lin et Stadtherr – pour produire des approximations polyhédrales fiables intérieure et extérieure de l'ensemble solution ainsi qu'une linéarisation de la fonction objectif. Enfin, d'autres ingrédients originaux font partie de notre optimiseur, comme un algorithme de propagation de contraintes sur intervalles exploitant la monotonie des fonctions.

Nous proposons au final un nouveau schéma fiable d'optimisation globale continue sous contraintes. Une implantation est disponible en tant qu'extension de l'outil `Ibex` (bibliothèque libre en C++ de résolution par intervalles). En termes de performances, notre stratégie dépasse de manière significative les meilleurs optimiseurs globaux fiables.

1 Introduction

Les algorithmes de B&B sur intervalles sont utilisés pour résoudre des problèmes d'optimisation globale sous contraintes[1] de manière fiable. Ils fournissent soit une solution optimale (et le coût associé avec une erreur bornée), soit une preuve d'infaisablité. Historiquement, le B&B sur intervalles est né avec l'analyse par intervalles [13]. Des travaux pionniers sont décrits dans [13], [7] ou [9]. Au milieu des années 1990, Kearfott a conçu le résolveur `GlobSol`. Parallèlement, des chercheurs en programmation par contraintes ont développé les résolveurs `Numerica` [21] et `Icos` [11], introduisant respectivement de la propagation de contraintes sur intervalles et des relaxations linéaires fiables. Plus récemment, la communauté de programmation mathématique a proposé un résolveur, appelé ici `IBBA+`, qui intègre de la propagation de contraintes et de l'arithmétique affine [16].

Pour concilier fiabilité et bonne performance, les intervalles font face principalement à deux difficultés.

Améliorer le majorant dans l'espace faisable

La *recherche locale* est l'approche la plus utilisée pour trouver un point *faisable*[2] (une solution satisfaisant les contraintes) qui améliore la borne courante pour la fonction objectif f. Cependant, pour assurer la faisabilité en cas de contraintes d'égalité, il est nécessaire, dès lors que l'espace exploré contient potentiellement des points infaisables, d'appliquer un processus de correction et de certification après chaque itération de la recherche locale. Ce processus est basé sur des techniques par intervalles coûteuses [11]. Cette correction rend du coup impossible, en termes de perfor-

[1] Nous considérons dans cet article des problèmes de minimisation.

[2] Une seconde approche consiste à chercher les points qui annulent le gradient de f. Le minimum de f est alors soit une solution de ce problème, soit sur la frontière du domaine. Malheureusement, la prise en compte des contraintes dans cette formulation (conditions de Kuhn-Tucker) aboutit souvent à une fonction d'agrégation conséquente et inadaptée aux calculs par intervalles [7].

mance, la compétition avec des résolveurs *non* fiables d'optimisation globale, comme `Baron` [19].

Dans cet article, nous proposons une approche radicalement différente où l'amélioration de la borne se fait en explorant des régions intérieures, c.-à-d. des régions où l'on prouve en amont que tous les points sont faisables. La notion de boîte intérieure a déjà été étudiée en programmation par contraintes, que ce soit pour le pavage de l'espace solution [6, 2] ou pour minimiser le nombre de contraintes numériques d'inégalité non satisfaites [17]. En revanche, cette approche n'a pas encore été exploitée dans un cadre général d'optimisation globale sous contraintes d'égalité et d'inégalité.

Calcul d'un minorant par une relaxation linéaire fiable

Tous les résolveurs existants calculent, à chaque nœud du B&B, une approximation convexe (en général polyhédrale) de l'espace solution. La meilleure solution de cette relaxation fournit un minorant du coût, ce dernier étant indispensable pour terminer la recherche. Rappelons que l'approximation extérieure doit contenir l'ensemble solution en dépit des erreurs de calcul dues aux nombres à virgule flottante. Or, la plupart des relaxations linéaires sont trop sophistiquées pour pouvoir être rendues fiables facilement. Des techniques spécifiques de reformulation-linéarisation (RLT) [18] sont présentées dans [9] et [11]. Elles introduisent de nouvelles variables, représentant les opérateurs de puissance et de produit, et définissent des contraintes linéaires entre ces variables. Ninin et al. utilisent, eux, l'arithmétique affine pour effectuer une linéarisation fiable de chaque opérateur [16]. Nous proposons, pour notre part, une linéarisation fiable basée sur l'évaluation de Taylor par intervalles, au premier ordre. La simplicité de cette relaxation nous a permis de mettre aussi au point une version duale pouvant extraire une région polyhédrale cette fois intérieure de l'ensemble solution et ainsi améliorer la borne courante.

1.1 Intervalles et optimisation globale sous contraintes

Un intervalle $[x_i] = [\underline{x_i}, \overline{x_i}]$ est l'ensemble des nombres réels x_i tels que $\underline{x_i} \leq x_i \leq \overline{x_i}$. $\mathbb{IR}$ représente l'ensemble de tous les intervalles. La taille ou largeur de $[x_i]$ est $w([x_i]) = \overline{x_i} - \underline{x_i}$. Une boîte $[x]$ est le produit cartésien des intervalles $[x_1] \times ... \times [x_i] \times ... \times [x_n]$. Sa largeur est définie par $\max_i w([x_i])$. $\text{Mid}([x])$ est le milieu de $[x]$. Un problème continu d'optimisation globale sous contraintes se définit ainsi :

Définition 1 (Optim globale sous contraintes) *Soient x un vecteur de variables $x = (x_1, ..., x_i, ...x_n)$ dans une boîte $[x]$, f une fonction à valeurs réelles $f : \mathbb{R}^n \to \mathbb{R}$, $g : \mathbb{R}^n \to \mathbb{R}^m$ et $h : \mathbb{R}^n \to \mathbb{R}^p$, des fonctions à valeurs vectorielles.*

Étant donné le système $S = (f, g, h, x, [x])$, le problème d'optimisation globale sous contraintes consiste à trouver :

$$\min_{x \in [x]} \{f(x) \ t.q. \ g(x) \leq 0 \wedge h(x) = 0\}.$$

f est la **fonction objectif***; g et h sont les* **contraintes** *d'inégalité et d'égalité. Un point est dit* **faisable** *s'il satisfait les contraintes.*

Notre algorithme d'optimisation extrait des *régions intérieures* dans les boîtes extérieures classiques.

Définition 2 *Soit un système $(f, g, \emptyset, x, [x]^{out})$ ne comprenant que des contraintes d'inégalité. Une* **région intérieure** *r^{in} est un sous-ensemble faisable de $[x]^{out}$, c.-à-d. $r^{in} \subset [x]^{out}$ et tous les points $x \in r^{in}$ satisfont $g(x) \leq 0$.*

Une région intérieure $[x]^{in}$ qui est une boîte est appelée **boîte intérieure**.

Un des opérateurs de notre stratégie effectue des *linéarisations intérieures* et extrait ainsi de l'espace faisable des *polytopes intérieurs*.

L'*arithmétique d'intervalles* [13] étend à $\mathbb{IR}$ les fonctions élémentaires sur $\mathbb{R}$. Par exemple, la somme d'intervalles $([x_1] + [x_2] = [\underline{x_1} + \underline{x_2}, \overline{x_1} + \overline{x_2}])$ contient l'image de la fonction somme sur ses arguments, et cette propriété d'inclusion définit ce que nous appelons une extension aux intervalles.

Définition 3 (Extension d'une fonction à $\mathbb{IR}$) *Soit une fonction $f : \mathbb{R}^n \to \mathbb{R}$.*
$[f] : \mathbb{IR}^n \to \mathbb{IR}$ est une **extension** *de f aux intervalles ssi :*

$$\forall [x] \in \mathbb{IR}^n \quad [f]([x]) \supseteq \{f(x), \ x \in [x]\},$$
$$\forall x \in \mathbb{R}^n \quad f(x) = [f](x).$$

Dans notre contexte, l'expression d'une fonction f est toujours la composition de fonctions élémentaires. L'**extension naturelle** $[f]_N$ est alors simplement la composition des opérateurs correspondants de l'arithmétique d'intervalles.

Les linéarisations extérieure et intérieure proposées dans cet article sont liées à l'**extension de Taylor** au premier ordre [13], définie comme suit :

$$[f]_T([x]) = f(\dot{x}) + \sum_i \left[\frac{\partial f}{\partial x_i}\right]_N ([x]) * ([x_i] - \dot{x_i})$$

où $\dot{x}$ est un point quelconque de $[x]$, par exemple, $\text{Mid}([x])$.

Exemple. Soit $f(x_1, x_2) = 3x_1^2 + x_2^2 + x_1 * x_2$ sur la boîte $[x] = [-1, 3] \times [-1, 5]$. L'évaluation naturelle donne : $[f]_N([x_1], [x_2]) = 3 * [-1, 3]^2 + [-1, 5]^2 + [-1, 3] * [-1, 5] = [0, 27] + [0, 25] + [-5, 15] = [-5, 67]$. Les dérivées partielles sont : $\frac{\partial f}{\partial x_1}(x_1, x_2) = 6x_1 + x_2$, $[\frac{\partial f}{\partial x_1}]_N([-1, 3], [-1, 5]) = [-7, 23]$, $\frac{\partial f}{\partial x_2}(x_1, x_2) = x_1 + 2x_2$, $[\frac{\partial f}{\partial x_2}]_N([x_1], [x_2]) = [-3, 13]$. L'évaluation de Taylor avec $\dot{x} = (1, 2)$ produit : $[f]_t([x_1], [x_2]) = 9 + [-7, 23] * [-2, 2] + [-3, 13] * [-3, 3] = [-76, 94]$.

1.2 Transformer les équations en inégalités

Pour traiter les égalités, une première option est suivie par la communauté d'analyse par intervalles. Elle consiste à trouver *approximativement* un point qui satisfait *exactement* les contraintes. Le résolveur renvoie une petite boîte de largeur ϵ_{sol} dans laquelle l'existence d'un point faisable à valeurs réelles est (souvent) garantie par des méthodes de type Newton par intervalles. Une seconde option consiste à trouver *exactement* un point qui satisfait *approximativement* les contraintes. Les équations sont traitées avec une erreur de précision admissible ϵ_{eq}, c'est-à-dire qu'un point faisable à virgule flottante x doit vérifier $h(x) \in [-\epsilon_{eq}, +\epsilon_{eq}]$. Toutes les contraintes peuvent alors être vues comme des inégalités : $\{g(x) \leq 0, h(x) - \epsilon_{eq} \leq 0, -h(x) - \epsilon_{eq} \leq 0\}$. Ninin et al. ont été guidés vers ce choix par l'arithmétique affine, mais nous pensons que c'est une approche pertinente pour tout résolveur d'optimisation globale. Remarquons tout d'abord que les deux approches ont un statut équivalent en termes de fiabilité. Ensuite, une précision ϵ_{eq} sur les *images* des fonctions h répond mieux au problème de la faisabilité qu'une précision ϵ_{sol} sur les inconnues. D'autre part, la plupart des équations définies par les utilisateurs sont déjà "épaisses" et n'ont pas besoin d'une relaxation additionnelle à ϵ_{eq} près. En effet, les contraintes contiennent souvent des coefficients connus avec une incertitude bornée (par exemple une imprécision dans une mesure) ou des constantes irrationnelles comme π qui ne peuvent être entrées que sous forme d'intervalles. Enfin, nos expérimentations ont mis en évidence que le traitement de ces égalités épaisses peut être efficace en pratique. La raison derrière cette bonne surprise est que cette approche permet aux résolveurs d'extraire des régions intérieures dans des continuums de solutions. *Les algorithmes d'extraction de régions intérieures et de contraction peuvent focaliser la recherche dans le petit espace solution défini par les équations épaisses.*

2 Notre B&B sur intervalles

Notre stratégie `IbexOpt` suit le schéma bien connu de séparation-évaluation (B&B) décrit dans [8] pour résoudre un problème d'optimisation globale sous contraintes. L'algorithme effectue récursivement, à partir d'une boîte initiale, des découpages jusqu'à obtenir une solution qui minimise la fonction objectif. Pendant la recherche, un minorant (généralement non faisable) de la fonction objectif est maintenu incrémentalement à partir de la liste des boîtes traitées ou en attente. Nous notons *lb* (pour *lower bound*) le plus petit de ces minorants. De même, *ub* (pour *upper bound*) désigne le coût du meilleur point faisable trouvé au cours de la recherche. Un critère d'arrêt est atteint quand $ub - lb$ est inférieur à la précision requise ϵ_{obj},[3] et le point flottant x_{ub} de coût *ub* est retourné. Notons que les boîtes dont la largeur est inférieure à la précision ϵ_{sol} ne sont pas remises dans la liste et que leurs minorants prennent part au calcul de *lb*.

A chaque itération, l'algorithme choisit dans la liste la boîte $[x]$ avec le plus petit minorant, réalisant ainsi une recherche en *meilleur d'abord*. La variable $x_i \in x$ est choisie par une heuristique, son domaine $[x_i]$ est bissecté et la procédure principale `Contract&Bound` est appliquée sur les deux sous-boîtes. On notera que l'arbre de recherche, c.-à-d. la "liste" des boîtes à traiter, est implanté par une structure de *tas* qui permet d'accéder au plus petit élément en temps constant. On trouvera plus de détails sur ce schéma dans [16].

La première tâche de notre algorithme est l'introduction d'une nouvelle variable y dans le système $(f, g, h, x, [x])$, à l'instar de Numerica [21] par exemple. Cette variable est liée aux autres par la contrainte supplémentaire $y = f(x)$. Son domaine $[y]$ est donc un intervalle qui contient l'image de la fonction objectif sur $[x]$. Cette extension du système permet de propager et rétro-propager automatiquement les contractions entre $[x]$ et les bornes globales sur le minimum. Ainsi, la boîte étendue $[x] \times [y]$ définit l'état courant de l'optimiseur. S'y ajoutent trois variables partagées par tous les nœuds de l'arbre de recherche et mises à jour de manière globale pendant la recherche : la meilleure solution courante x_{ub}, son coût *ub* ($f(x_{ub}) = ub$) et *lb* le minimum des minorants des boîtes non traitées.

2.1 Stratégie de bissection

A chaque nœud de l'arbre de recherche, une variante de l'heuristique bien connue de la *smear function* [10] permet de choisir la prochaine variable à bissecter. Étant donné un système $(f, g, h, x, [x])$, la stratégie *smear* standard choisit la variable x_i de x qui maximise

[3] Conformément aux implantations standard, ϵ_{obj} est un pourcentage de *ub* si $|ub| \geq 1$ et une distance absolue si $|ub| \leq 1$.

suivant les cas $\texttt{smearMax}(x_i) = Max_{f_j}\ \texttt{smear}(x_i, f_j)$ ou $\texttt{smearSum}(x_i) = \sum_{f_j} \texttt{smear}(x_i, f_j)$, où f_j représente soit la fonction objectif f, soit une contrainte de g ou h.

$\texttt{smear}(x_i, f_j)$ est une mesure de l'impact de la variable x_i sur la fonction f_j. Son calcul implique la dérivée partielle de f_j par rapport à x_i et la largeur de $[x_i]$. Plus précisément :

$$\texttt{smear}(x_i, f_j) = \left| \left[\frac{\partial f_j}{\partial x_i} \right]_N ([x]) \right| * w([x_i]).$$

Nous proposons une variante, $\texttt{smearRel}(x_i, f_j)$, qui mesure un impact *relatif*, à valeur dans $[0, 1]$:

$$\texttt{smearRel}(x_i, f_j) = \frac{\texttt{smear}(x_i, f_j)}{\sum_{x_k \in x} \texttt{smear}(x_k, f_j)}.$$

Finalement, notre stratégie de bissection SmearSumRel choisit la variable x_i de x avec le plus grand impact :

$$\texttt{smearSumRel}(x_i) = \sum_{f_j} \texttt{smearRel}(x_i, f_j).$$

Même si elle n'est pas toujours la meilleure, cette stratégie apparaît plus robuste que ses concurrentes sur l'ensemble des problèmes testés.

2.2 La procédure Contract&Bound

L'algorithme principal Contract&Bound (cf. algorithme 1) est appelé à chaque nœud de notre B&B. La première ligne introduit dans le système courant le meilleur coût ub trouvé (comme dans tout B&B). La procédure OuterContractLB contracte les domaines et améliore le minorant. InnerExtractUB extrait une région intérieure dans $[x]$, prélève en cas de succès un point x dans cette région et, si x améliore le critère, remplace x_{ub} par x et le coût ub par $f(x)$.

Algorithm 1 Contract&Bound (**in** S, $[x]$; **in-out**: ub)

```
ȳ ← ub − ε_obj
OuterContractLB (S, [x] × [y])  /* contraction */
if [x] × [y] = ∅ then exit endif  /* no solution */
InnerExtractUB (S, [x], ub, x_ub) /* inner regions */
```

OuterContractLB appelle principalement deux procédures. La première est l'algorithme Mohc [1] qui contracte la boîte $[x]\times[y]$ en donnant, par effet de bord, un minorant à la fonction objectif. Cet algorithme récent de propagation de contraintes sur intervalles exploite la monotonie des fonctions. Il utilise une procédure *Revise* efficace qui contracte de manière optimale la boîte par rapport à une contrainte (par exemple, $g_j(x) \leq 0$) si $g_j(x)$ est détectée comme étant monotone suivant toutes les variables et en tout point de la boîte, même si $g_j(x)$ contient des occurrences multiples de variables. On notera que plus la boîte est petite (ou de façon équivalente, plus on descend en profondeur dans l'arbre de recherche), plus les fonctions deviennent monotones.

La seconde procédure appelée par OuterContractLB est une relaxation linéaire, nommée ici OuterLinearization, qui calcule un minorant de la fonction objectif et met à jour $\underline{y}$.

2.3 Relaxation linéaire extérieure

La relaxation linéaire ci-dessous est une adaptation directe de la forme de Taylor du premier ordre sur intervalles [12]. Soit une fonction $f : \mathbb{R}^n \to \mathbb{R}$ définie sur une boîte $[x]$. Pour toute variable $x_i \in x$, soit $[a_i] : \left[\frac{\partial f}{\partial x_i}\right]_N([x])$. Le principe consiste à minorer $f(x)$ par des fonctions linéaires :

$$\forall x \in [x], f(\underline{x}) + \underline{a_1} * x_1^l + ... + \underline{a_n} * x_n^l \ \leq f(x) \quad (1)$$

$$\forall x \in [x], f(\overline{x}) + \overline{a_1} * x_1^r + ... + \overline{a_n} * x_n^r \ \leq f(x) \quad (2)$$

avec : $x_i^l = x_i - \underline{x_i}$ et $x_i^r = x_i - \overline{x_i}$.

La forme de Taylor au premier ordre sur intervalles peut s'appliquer avec n'importe quel point $\dot{x}$ à l'intérieur de la boîte. Au lieu du milieu, choisi habituellement, nous prenons ici un *coin* de la boîte : $\underline{x}$ dans la formule (1) ou $\overline{x}$ dans la formule (2). Si nous considérons une inégalité $g_j(x) \leq 0$, l'expression (1) (ou (2)) définit ainsi un hyperplan $g_j^l(x)$ bornant inférieurement l'espace solution : $g_j^l(x) \leq g_j(x) \leq 0$. En appliquant, par exemple, la formule (1) à la fonction objectif $f(x)$ et aux inégalités $g_j(x) \leq 0$ $(j = 1...m)$, on peut générer un problème linéaire LP^{lb} qui est une relaxation du problème initial :

$$\begin{array}{lll} LP^{lb} = \min & & f(\underline{x}) + \underline{a_1} * x_1^l + ... + \underline{a_n} * x_n^l \\ sous : & \forall j & g_j(\underline{x}) + \underline{a_1^j} * x_1^l + ... + \underline{a_n^j} * x_n^l \leq 0 \\ & \forall i & 0 \leq x_i^l, \quad x_i^l \leq w([x_i]) \\ avec : & & x_i^l = x_i - \underline{x_i} \end{array}$$

OuterLinearization invoque ensuite l'algorithme du simplexe pour résoudre LP^{lb}. Il calcule la valeur optimale y^l ou détecte une infaisabilité. L'infaisabilité indique que $[x]$ ne contient pas de solution et peut être éliminée. Si au contraire $y^l \geq \underline{y}$, alors le minorant de l'objectif sur la boîte est mis à jour : $\underline{y} \leftarrow y^l$.

Proposition 1 *Les linéarisations par intervalles (1) et (2) sont correctes et fiables, c.-à-d., elles peuvent être rendues robustes par rapport aux erreurs d'arrondis sur les nombres à virgule flottante.*

La fiabilité est assurée par la taylorisation sur intervalles [14]. La correction de la relation (1) repose sur le choix d'un coin comme point d'expansion. Elle tient au fait que toute variable x_i^l est positive puisque son domaine est $[0, d_i]$, avec $d_i = w([x_i]) = \overline{x_i} - \underline{x_i}$. Ainsi, le minimum de chaque terme $[a_i] * x_i^l$ pour un point $x_i^l \in [0, d_i]$ est obtenu avec $\underline{a_i}$. Symétriquement, la relation (2) est correcte puisque $x_i^r \in [-d_i, 0] \leq 0$, et le minimum de chaque terme est obtenu avec $\overline{a_i}$ [12].

Il faut noter que, bien que nos linéarisations soient fiables, les erreurs de calcul dus aux nombres flottants dans l'algorithme du simplexe peuvent rendre ses résultats non fiables. Pour rendre la solution du simplexe fiable, nous avons ajouté un post-traitement peu coûteux, proposé dans [15], utilisant l'arithmétique d'intervalles.

Nous avons apporté une amélioration à notre relaxation linéaire pour calculer un polytope plus petit. Nous minorons une fonction $f(x)$ avec les expressions (1) et (2) simultanément, en utilisant une forme développée :

1. $f(\underline{x}) + \sum_i \underline{a_i}(x_i - \underline{x_i}) = f(\underline{x}) + \sum_i (\underline{a_i} x_i - \underline{a_i}\,\underline{x_i})$
   $= \sum_i \underline{a_i} x_i + f(\underline{x}) - \sum_i \underline{a_i}\,\underline{x_i}$
2. $f(\overline{x}) + \sum_i \overline{a_i}(x_i - \overline{x_i}) = f(\overline{x}) + \sum_i (\overline{a_i} x_i - \overline{a_i}\,\overline{x_i})$
   $= \sum_i \overline{a_i} x_i + f(\overline{x}) - \sum_i \overline{a_i}\,\overline{x_i}$

2.4 Trouver des majorants dans des régions intérieures

L'appel à `OuterContractLB` est suivi par un appel à `InnerExtractUB` (cf. algorithme 2). La procédure commence par appeler une adaptation d'un algorithme récent, nommé ici `InHC4` [4], pour extraire une boîte intérieure à partir de la boîte extérieure $[x]^{out}$.[4] Appliqué à une contrainte, `InHC4` renvoie une boîte intérieure pour cette contrainte. Les différentes boîtes retournées pour les différentes contraintes sont intersectées pour obtenir une boîte intérieure. Comme `HC4` [3], l'algorithme raisonne sur l'arbre syntaxique des expressions et utilise des *projections* pour les opérateurs unaires, avec arrondi vers l'intérieur. De plus, dans le cas d'unions disjointes d'intervalles (par exemple, pour les opérateurs x^2 et *sinus*), on ne garde qu'un seul intervalle puisque les trous contiennent des points incohérents, ce qui rend l'algorithme heuristique. Pour les opérateurs binaires, les projections dans la phase de rétro-propagation (*projection*) sont différentes et conduisent aussi à des choix heuristiques. Pour plus de détails, se référer à la section 3 de [4].

Si `InHC4` trouve une boîte intérieure $[x]^{in}$, alors `MonotonicityAnalysis` analyse la monotonie de la fonction objectif par rapport à chaque variable x_i. Si la dérivée partielle $[a_i] = \left[\frac{\partial f}{\partial x_i}\right]_N([x]^{in}) \geq 0$, alors f est croissante et $[x_i]$ est remplacé par l'intervalle dégénéré $[\underline{x_i}, \underline{x_i}]$ dans $[x]^{in}$ pour minimiser $f(x)$ dans $[x]^{in}$. Si $[a_i] \leq 0$, f est décroissante et $[x_i]$ est remplacé par $[\overline{x_i}, \overline{x_i}]$ dans $[x]^{in}$.

Algorithm 2 `InnerExtractUB` (**in :** S, $[x]^{out}$; **in-out :** ub, x_{ub})

```
[x]^in ← InHC4 (S, [x]^out)   /* Inner box extraction */
if [x]^in ≠ ∅ then
  [x]^in ← MonotonicityAnalysis (f, [x]^in)
  x ← RandomProbing([x]^in)
else
  x ← RandomProbing([x]^out)
end if
cost ← ⌈[f]_N([x,x])⌉   /* Cost evaluation */
if cost < ub and ([x]^in ≠ ∅ or ⌈[g]_N([x,x])⌉ ≤ 0)
then
  ub ← cost ; x_ub ← x
end if
LP^ub ← InnerLinearization (S, [x]^out)
x^l ← Simplex(LP^ub)
if x^l ≠ ⊥ then
  cost ← ⌈[f]_N([x^l, x^l])⌉
  if cost < ub then ub ← cost ; x_ub ← x^l end if
end if
```

Ensuite, nous prenons un point aléatoirement dans la boîte[5] et remplaçons x_{ub} par x si x satisfait les contraintes et améliore le meilleur coût ub. Deux cas différents peuvent se produire. Si `InHC4` a extrait une boîte intérieure $[x]^{in}$, on prend alors un point dans $[x]^{in}$ sans avoir besoin de tester la faisabilité puisque $[x]^{in}$ ne contient que des points faisables. Si aucune boîte intérieure n'a été trouvée, un point est choisi au hasard dans la boîte extérieure et les contraintes doivent être vérifiées. Le remplacement de ce simple tirage aléatoire par une descente de gradient n'apporte pas d'amélioration en pratique. Cela s'explique facilement en présence d'équations puisque les boîtes intérieures sont très petites. C'est en revanche plus surprenant pour les problèmes d'optimisation ne contenant que des contraintes d'inégalité.

La dernière étape de `InnerExtractUB` consiste à linéariser le système pour en extraire cette fois une région polyhédrale intérieure.

2.5 Linéarisation intérieure par intervalles

De manière symétrique à la relation (1) utilisée dans la linéarisation extérieure, on a pour la linéarisation

[4] L'algorithme publié traite en fait un problème dual de recherche de boîtes infaisables, c.-à-d. des boîtes où tous les points satisfont la négation d'au moins une contrainte.

[5] Sélectionner plusieurs points au lieu d'un seul s'est avéré contre-productif dans nos expérimentations.

intérieure :

$$\forall x \in [x],\ f(x) \leq f^l(x) = f(\underline{x}) + \sum_i \overline{a_i} * (x_i - \underline{x_i}). \quad (3)$$

Si on traite une inégalité $f(x) \leq 0$, la relation (3) permet de construire un hyperplan $f^l(x)$ tel que $f(x) \leq f^l(x) \leq 0$. Cette fonction linéaire $f^l(x)$ peut donc être utilisée pour décrire une région intérieure de $[x]$. En appliquant cette idée à la fonction objectif $f(x)$ et aux inégalités $g_j(x) \leq 0$, on peut en déduire le programme linéaire LP^{ub} suivant :

$$\begin{array}{lll} LP^{ub} = \min & & f(\underline{x}) + \sum_i \overline{a_i} * (x_i - \underline{x_i}) \\ sous: & \forall j & g_j(\underline{x}) + \sum_i \overline{a_i^j} * (x_i - \underline{x_i}) \leq 0 \\ & \forall i & \underline{x_i} \leq x_i \wedge x_i \leq \overline{x_i} \end{array}$$

De nouveau, l'algorithme du simplexe résout LP^{ub} et retourne la solution optimale x^l ou lève une infaisabilité (cf. algorithme 2). L'infaisabilité cette fois ne prouve rien car le système linéarisé est plus contraint que le système original. Si l'algorithme du simplexe retourne une solution optimale de l'approximation intérieure, alors x^l est aussi une solution du système original, mais généralement pas la solution optimale. On évalue donc la fonction objectif (originale) au point x^l, celle-ci devant être inférieure à ub pour pouvoir mettre à jour x_{ub} et ub.

3 Expérimentations

Nous avons implanté notre stratégie dans le logiciel libre `Ibex` (Interval-Based EXplorer) [5]. Cette bibliothèque a facilité l'implantation de notre optimiseur global en fournissant un certain nombre de briques telles que l'algorithme `Mohc`, diverses stratégies de branchement, la dérivation automatique, etc. Tous les paramètres ont été fixés à un ensemble de valeurs communes à toutes les instances testées. La précision a été fixée à $\epsilon_{obj} =$ `1.e-8` et $\epsilon_{sol} = \frac{\epsilon_{obj}}{10}$. Enfin, l'erreur admissible ϵ_{eq} sur les équations épaisses $h(x) \in [-\epsilon_{eq}, +\epsilon_{eq}]$ a été fixée à $\epsilon_{eq} =$`1.e-8` dans toutes les expérimentations.

Les expérimentations ont été réalisées sur l'ensemble des 74 systèmes de la base de problèmes Coconut[6] sélectionnés par notre meilleur compétiteur fiable `IBBA+` [16]. Le tableau 1 présente une étude qualitative analysant quels sont les ingrédients qui améliorent la performance.

On peut faire quelques observations intéressantes. Tout d'abord, les cinq ingrédients originaux intégrés à notre stratégie s'avèrent tous utiles en pratique. En particulier, la linéarisation extérieure simple que nous avons proposée est souvent cruciale pour la phase de calcul d'un minorant. Des travaux futurs devront comparer cette taylorisation convexe sur intervalles avec l'arithmétique affine et l'opérateur de RLT `Quad` utilisé dans `Icos`. Enfin, l'extraction de régions intérieures est aussi très utile dans la phase de recherche d'un majorant faisable. Le tableau 1 souligne qu'il suffit souvent de réaliser cette extraction de boîte intérieure avec `InHC4` *ou* `InnerLinearization`, mais que leur introduction conjointe est parfois bénéfique et jamais contre-productive sur l'ensemble des problèmes testés.

TAB. 1 – **Étude qualitative.** Les colonnes indiquent le nombre de systèmes dont la perte en performance $\frac{temps\,CPU(stratégie \backslash \{ingrédient\})}{temps\,CPU(stratégie)}$, causée par le retrait d'un seul ingrédient de notre stratégie, appartient à un intervalle donné (première ligne). Les retraits testés sont : `Mohc` remplacé par `HC4` (Mohc/HC4) ; `OuterLinearization` (OuterLinear.) ; `InnerExtractUB` remplacé par un simple tirage aléatoire dans la boîte extérieure (Inner/Probing) ; `InnerLinearization` (InnerLinear.) ; `InHC4` ; `SmearSumRel` remplacé resp. par `SmearMax` (SSR/SM) ; `Round Robin` (SSR/RR) ; `LF` (SSR/LF) ; l'heuristique `LargestFirst` choisit la variable avec l'intervalle le plus large.

Gain	0.02	[0.1, 0.5]	[0.5, 2]	[2, 10]	[10, 100]	>100
Mohc/HC4	0	1	62	5	0	2
OuterLinear.	0	1	35	9	5	20
Inner/Probing	0	0	33	24	9	4
InnerLinear.	0	0	62	7	0	1
InHC4	0	0	66	4	0	0
SSR/SM	0	2	59	4	1	4
SSR/RR	0	1	42	13	11	3
SSR/LF	1	0	40	9	16	4

Nous avons aussi comparé notre stratégie avec des résolveurs d'optimisation globale fiables et disponibles, `Globsol`, `Icos` et `IBBA+`[7], ainsi qu'avec le résolveur complet mais *non fiable* `Baron`. Remarquons que `Baron` ne garantit pas la solution renvoyée qui peut parfois être non faisable et avoir un coût trop bas.

La figure 1 montre les profils de performance de `IbexOpt`, `Baron` et de notre meilleur concurrent fiable `IBBA+`.

Nous donnons également des résultats détaillés sur les 25 systèmes qui sont résolus par `IbexOpt` en plus d'une seconde. Le tableau 2 correspond aux 12 systèmes résolus par `IbexOpt` entre 1 et 10 secondes. Le tableau 3 contient 13 systèmes résolus en plus de 10 secondes. Trois systèmes (`ex6_2_5`, `ex6_2_7` et `ex7_2_3`) ne sont résolus par aucun résolveur, y compris `Baron`.

[6] `www.mat.univie.ac.at/~neum/glopt/coconut/Benchmark/`

[7] `IBBA+` correspond à la stratégie la plus efficace dans [16].

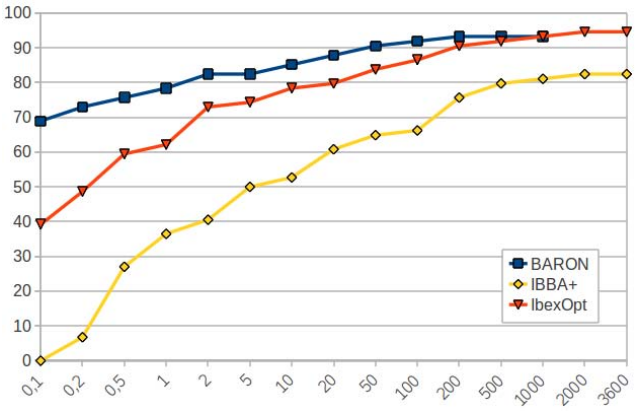


FIG. 1 – **Profils de performance.** Pour un algorithme donné, un point (t, p) sur la courbe correspondante indique le pourcentage p de systèmes résolus en moins de t secondes.

Les résultats pour `Globsol`, `IBBA+`, `Icos` et `IbexOpt` ont été obtenus sur des processeurs similaires (`Intel X86, 3Ghz`). `Baron` 9.0.7 a été lancé sur le serveur `Neos` (cf. `www.neos-server.org/neos/`), également un processeur `X86`, ce qui rend la comparaison relativement équitable.

Les profils de performance et les tableaux montrent que `IbexOpt` dépasse souvent ses concurrents fiables d'un ou plusieurs ordres de grandeur. Les profils illustrent en particulier le fait que `IbexOpt` a des performances intermédiaires entre `IBBA+` et `Baron` et qu'il peut résoudre les mêmes systèmes que `Baron` en 1000 secondes (540 s en fait). Les résultats obtenus par `Baron` sont impressionnants, même si l'on peut noter que plusieurs instances sont résolues durant un prétraitement (le nombre de nœuds est 1).

On notera aussi que `IbexOpt` est meilleur que `Baron` sur 6 des 25 systèmes difficiles (voir tableaux 2 et 3), spécialement sur la série `ex6_2_*` de problèmes qui ont des fonctions objectifs hautement non polynomiales. A notre connaissance, aucun autre résolveur fiable n'est compétitif avec `Baron` sur des instances non triviales que `Baron` résout en plus d'une seconde.

Nous avons aussi testé une variante de notre stratégie où `Mohc` est remplacé par `3BCID(Mohc)` [20]. Bien que généralement contre-productive en termes de performances, cette variante est plus robuste et peut résoudre l'instance `ex7_2_3` en 38 secondes et 6235 branchements, tandis que `Baron` explose en mémoire.

4 Conclusion

Nous avons proposé un nouveau cadre pour l'optimisation globale fiable qui exploite des régions intérieures dans la phase de recherche de majorant faisable, évitant ainsi le recours à la recherche locale. En définissant les équations avec une petite erreur admissible, cette approche permet aussi de traiter les contraintes d'égalité. Notre stratégie comprend cinq ingrédients utiles. Trois d'entre eux, `Mohc`, `InHC4` et `OuterLinearization` n'avaient jamais été utilisés en optimisation globale. Deux d'entre eux, `SmearSumRel` et `InnerLinearization` sont nouveaux. Tous les cinq ont montré leur efficacité sur un ensemble de problèmes d'optimisation globale non triviaux. Ils confirment la pertinence de l'exploitation des régions intérieures et des approximations polyhédrales basées sur une taylorisation convexe sur intervalles.

Ce nombre de nouveaux ingrédients laisse un espace significatif à des améliorations futures avec l'espoir d'atteindre à long terme les performances de `Baron`.

TAB. 2 – **Comparaison sur les systèmes de difficulté moyenne.** Les deux premières lignes indiquent le nom de chaque compétiteur avec la précision ϵ_{obj} sur le coût. Chaque entrée contient généralement le temps CPU en secondes (première ligne d'une multi-ligne) et le nombre de branchements (deuxième ligne). Une limite de temps de 1 heure (>3600) est commune à `IBBA+`, `GlobSol` et `IbexOpt`. Elle est de 10 mn (>600) pour `Icos`, 1000 secondes pour `Baron` (imposée par le serveur `Neos`). Une case vide indique que l'information n'est pas disponible. En particulier, `GlobSol` s'est restreint à des problèmes ayant moins de 9 variables.

Système ϵ_{obj}	n	Baron 1.e-8	GlobSol 1.e-8	IBBA+ 1.e-8	Icos 1.e-3	IbexOpt 1.e-3	IbexOpt 1.e-8
ex2_1_7	20	0.33 89		16.75 1574	>600	5.52 2102	6.24 2320
ex2_1_8	24	0.07 7		26.78 1916	>600	5.78 1540	6.50 1702
ex3_1_1	8	0.51 453	>3600	116 131195	180 8930	0.48 605	1.31 1516
ex6_1_4	6	0.25 242	14	2.70 1622	4.28 1109	0.37 471	1.11 1053
ex6_2_14	4	**5.2** 1824	32	208 95170	>600	0.77 765	**1.59** 1237
ex7_2_1	7	0.05 1		24.72 8419	>600	0.80 825	1.17 1197
ex7_2_6	3	0.06 7	1	1.23 1319	2.68 986	0.02 73	5.35 16171
ex7_3_4	12	0.93 268		>3600	>600	1.27 771	1.31 775
ex14_2_1	5	0.03 1	4	36.73 16786	>600	0.82 533	1.09 704
ex14_2_3	6	0.03 1	11	173 46673	>600	2.57 996	2.92 1048
ex14_2_4	5	0.03 1		127 30002	>600	0.95 435	1.02 449
ex14_2_6	5	0.03 1		237 74630	>600	1.20 498	1.29 515

TAB. 3 – **Comparaison sur les systèmes difficiles.** En cas de limite de temps atteinte par `Baron` ou `IbexOpt`, la seconde ligne indique la précision obtenue.

Système ϵ_{obj}	n	Baron 1.e-8	GlobSol 1.e-8	IBBA+ 1.e-8	Icos 1.e-3	IbexOpt 1.e-3	IbexOpt 1.e-8
ex2_1_9	10	1.52 2050		154 60007	59.9 1549	13 13370	30 30444
ex6_1_1	8	7.64 5616	3203	>3600	>600	13 12811	17 14725
ex6_1_3	12	19.2 11217		>3600	>600	46.74 26137	540 204439
ex6_2_6	3	26 26765	306	1575 922664	>600	36.75 34318	173 163227
ex6_2_8	3	19 29469	220	458 265276	>600	29.40 27513	111 97554
ex6_2_9	4	**170** 92143	465	522 203775	>600	12.94 9873	**37** 27461
ex6_2_10	6	**>1000** **2.e-3**	>3600	>3600	>600	**431** 224484	1955 820902
ex6_2_11	3	**55** 45085	273	140 83457	>600	4.02 4487	**22** 24264
ex6_2_12	4	30 19182	193	113 58231	>600	4.37 4173	122 86722
ex6_2_13	6	**>1000** **2.e-2**	>3600	>3600	>600	**1099** 545676	>3600 2.e-4
ex7_3_5	13	1.11 309		>3600	136 3699	50.50 40936	55 44147
ex14_1_7	10	1.27 181		>3600	>600	451 177464	464 181136
ex14_2_7	6	0.03 1		>3600	>600	84.73 17463	85 16759

Références

[1] I. Araya, G. Trombettoni, and B. Neveu. Exploiting Monotonicity in Interval Constraint Propagation. In *Proc. AAAI*, pages 9–14, 2010.

[2] F. Benhamou and F. Goualard. Universally Quantified Interval Constraints. In *Proc. CP*, pages 67–82, 2000.

[3] F. Benhamou, F. Goualard, L. Granvilliers, and J.-F. Puget. Revising Hull and Box Consistency. In *Proc. ICLP*, pages 230–244, 1999.

[4] G. Chabert and N. Beldiceanu. Sweeping with Continuous Domains. In *Proc. CP, LNCS 6308*, pages 137–151, 2010.

[5] G. Chabert and L. Jaulin. Contractor Programming. *Artificial Intelligence*, 173 :1079–1100, 2009.

[6] H. Collavizza, F. Delobel, and M. Rueher. Extending Consistent Domains of NCSP. In *IJCAI*, pages 406–413, 1999.

[7] E. R. Hansen. *Global Optimization Using Interval Analysis*. Marcel Dekker Inc., 2003.

[8] R. Horst and H. Tuy. *Global Optimization : Deterministic Approaches*. Springer, 1966.

[9] R. B. Kearfott. *Rigourous Global Search : Continuous Problems*. Kluwer Academic Publishers, 1996.

[10] R.B. Kearfott and M. Novoa III. INTBIS, a portable interval Newton/Bisection package. *ACM Trans. on Mathematical Software*, 16(2) :152–157, 1990.

[11] Y. Lebbah, C. Michel, and M. Rueher. An Efficient and Safe Framework for Solving Optimization Problems. *J. of Computational and Applied Mathematics*, 199 :372–377, 2007.

[12] Y. Lin and M. Stadtherr. LP Strategy for the Interval-Newton Method in Deterministic Global Optimization. *Ind. & eng. chemistry research*, 43 :3741–3749, 2004.

[13] R. E. Moore. *Interval Analysis*. Prentice-Hall, 1966.

[14] A. Neumaier. *Interval Methods for Systems of Equations*. Cambridge Univ. Press, 1990.

[15] A. Neumaier and O. Shcherbina. Safe Bounds in Linear and Mixed-Integer Programming. *Mathematical Programming*, 99 :283–296, 2004.

[16] J. Ninin, F. Messine, and P. Hansen. A Reliable Affine Relaxation Method for Global Optimization. *Mathematical Programming, accepted for publication*, 2011.

[17] J.-M. Normand, A. Goldsztejn, M. Christie, and F. Benhamou. A Branch and Bound Algorithm for Numerical Max-CSP. *Constraints*, 15(2) :213–237, 2010.

[18] H. Sherali and W. Adams. *Reformulation-Linearization Technique for Solving Discrete and Continuous Nonconvex Problems*. Kluwer Academic Publishers, 1999.

[19] M. Tawarmalani and N. V. Sahinidis. A Polyhedral Branch-and-Cut Approach to Global Optimization. *Mathematical Programming*, 103(2) :225–249, 2005.

[20] G. Trombettoni and G. Chabert. Constructive Interval Disjunction. In *Proc. CP, LNCS 4741*, pages 635–650, 2007.

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

Sélection autonome d'opérateurs par dominance pour la recherche locale

Nadarajen Veerapen Frédéric Saubion
LUNAM Université, Université d'Angers, LERIA
{prenom.nom}@univ-angers.fr

Résumé

Cet article présente l'étude d'une méthode de sélection d'opérateurs pour un processus de recherche locale. L'objectif principal est de proposer une méthode générique de recherche locale qui gère la sélection d'opérateurs à partir d'un ensemble d'opérateurs disponibles. Ces opérateurs sont construits à partir de relations de voisinages et de fonctions de sélection de voisin. La sélection d'opérateurs utilise le concept de Pareto dominance en se basant sur des mesures de qualité et de distance. Cette dernière est déterminée à l'aide de deux métriques. L'algorithme de contrôle est codé en langage COMET afin d'être facilement utilisable dans d'autres algorithmes de recherche locale. L'emphase étant mise sur les problèmes de permutation, nous présentons des résultats expérimentaux pour le QAP et le ATSP afin d'évaluer l'efficacité de la méthode.

Abstract

This paper presents a study for the dynamic selection of operators in a local search process. The main purpose is to propose a generic autonomous local search method which manages operator selection from a set of available operators, built on neighborhood relations and neighbor selection functions, using the concept of Pareto dominance with respect to quality and diversity. The latter is measured using two different metrics. This control method is implemented using the COMET language in order to be easily introduced in various constraint local search algorithms. Focusing on permutation-based problems, experimental results are provided for the QAP and ATSP to assess the method's effectiveness.

1 Introduction

Les métaheuristiques ont largement été adoptées comme méthodes de résolution efficaces pour l'optimisation combinatoire et les problèmes de satisfaction de contraintes. Néanmoins, ces approches demandent souvent une expertise conséquente aussi bien du problème que de la méthode de résolution. Une approche récente, la construction de stratégies de contrôle haut niveau, a pour but de rendre les techniques d'optimisation plus facile à utiliser [4].

Du point de vue de la recherche locale (LS), un bon algorithme LS [10] doit explorer l'espace de recherche de façon efficace en quête de la solution optimale. Cela implique de jongler entre deux objectifs généralement divergents : l'intensification (converger vers un optimum local) et la diversification (échantillonner convenablement les différentes parties de l'espace de recherche). L'efficacité de ces deux stratégies dépend largement de la (des) structure(s) de voisinage choisie(s). Cet équilibre peut être contrôlé par le biais d'opérations de base (des mouvements) qui sont appliquées au cours du processus de recherche. De ce fait, un nombre croissant de travaux essaient de construire des algorithmes plus autonomes [8]. Cette tendance a été explorée dans le contexte de la recherche réactive [1], basée sur des travaux fondateurs comme le tabou réactif [2] ou le recuit simulé adaptatif [13]. Le concept général d'un changement systématique de relation de voisinage, basé sur le fait qu'un optimum est défini en fonction d'un voisinage et donc que différents voisinages n'auront pas forcément les mêmes optima, a été développé dans la recherche à voisinage variable (VNS) [9]. Il en existe des versions adaptatives, par exemple [11]. Néanmoins, comme récemment mentionné dans [15], la plupart des algorithmes LS gèrent la diversité et la qualité comme deux objectifs opposés et utilisent donc des étapes alternées de diversification et d'intensification. À titre d'exemple, la recherche locale itérée (ILS) [17] est une succession alternée d'une perturbation et d'une recherche locale (une exploration complète d'un voisinage ou une LS

Cet article est une traduction de l'article *Pareto Autonomous Local Search* [27] présenté à LION5, Rome, 2011.

plus complexe). Les algorithmes LS se focalisent le plus souvent sur la qualité de la solution la plus récente, mais peuvent utiliser des mécanismes de prohibition pour éviter le piège des optima locaux. En accord avec les remarques de [15], nous estimons qu'une meilleure coordination peut être atteinte entre les deux objectifs, qualité et diversité, pouvant être évaluée à travers un compromis fluctuant en réponse à l'état du processus de recherche.

Des travaux récents sur les algorithmes évolutionnaires proposent de nouvelles techniques pour la sélection adaptative d'opérateurs. *Compass* [18] évalue la performance d'un opérateur à travers la scalairisation de l'amélioration de la qualité du parent vers l'enfant, la variation en diversité et le temps d'exécution. Dans [6], un *Bandit Manchot Multi-Bras* est utilisé pour sélectionner l'opérateur qui maximise la somme de deux valeurs, la première représentant la performance de l'opérateur et la seconde assurant que l'opérateur soit sélectionné un nombre infini de fois. En utilisant les principes de *Compass*, un algorithme adaptatif de recherche locale a été présenté dans [22]. Dans ces travaux la performance d'un opérateur est définie par rapport à un compromis statique entre la qualité et la diversité.

Dans cet article nous présentons d'abord un modèle algorithmique général pour la recherche locale définie comme étant un processus de sélection d'opérateurs de mouvement à partir d'un ensemble d'opérateurs disponibles, qui sont une combinaison d'une relation de voisinage et d'une sélection de voisin au sein de ce voisinage. Ensuite l'objectif de l'algorithme est de choisir et d'appliquer un opérateur à la solution actuelle afin de construire progressivement le chemin de recherche. Notre approche est donc double : 1) introduire un nouveau compromis entre qualité et diversité dans la recherche et 2) offrir un framework de contrôle pouvant utiliser des opérateurs génériques afin de résoudre une large palette de problèmes avec comme objectif d'offrir des solutions d'optimisation à des utilisateurs non-experts en les soulageant du travail de conception et de réglage d'algorithmes.

Nous présentons des fonctionnalités de contrôle pour la résolution de problèmes de permutation, c.-à-d. les problèmes dont la configuration peut être modélisée par des permutations. Ce cadre général nous permet de définir différents opérateurs en combinant des voisinages de permutations et des sélecteurs basiques.

À chaque étape de la recherche, les opérateurs sont sélectionnés d'après le principe de Pareto dominance, calculée par rapport aux performances enregistrées de chaque opérateur en termes d'intensification et de diversification. De plus, comme notre objectif est d'offrir un cadre de développement générique aux utilisateurs de recherche locale, nos fonctionnalités de contrôles sont écrites en COMET [26], un langage dédié à la conception d'algorithmes de recherche locale et intégrant la gestion des contraintes. Afin de souligner la généralité de notre contrôleur, nous le testons ensuite sur deux problèmes de permutation bien connus : le problème d'affectation quadratique et le problème du voyageur de commerce asymétrique.

Le reste de l'article est organisé en quatre sections. La section 2 pose les définitions permettant de manipuler les voisinages, les sélecteurs et les opérateurs pour les problèmes de permutation. Dans la section 3 nous présentons le cadre du contrôle, deux mesures de distance et la méthode de sélection Pareto. Ceci est suivi par le protocole de test et les résultats à la section 4. Enfin, la section 5 clos l'article par la conclusion ainsi que quelques possibilités de recherches futures.

2 Voisinages, sélecteurs et opérateurs

L'objectif de cette section est de présenter une description formelle des problèmes basés sur les permutations ainsi que les opérateurs qui leur sont associés. Dans [23], les auteurs considèrent formellement différents voisinages et définissent des distances associées à ces voisinages. Comme mentionné ci-dessus, puisque notre but est de gérer dynamiquement les opérateurs en fonction de leur comportement et de leurs propriétés, ces métriques nous intéressent tout particulièrement. Néanmoins, dans [23], les auteurs ne considèrent que des méthodes utilisant un unique opérateur et les métriques pouvant être utilisées pour évaluer la diversité de la trajectoire de la recherche locale dépendent entièrement de l'opérateur.

Ici nous avons pour objectif d'offrir une description simple et générique de ce que sont les voisinages et les opérateurs qui peuvent être utiles pour définir de nouveaux opérateurs et gérer leur application en fonction de leur impact sur le processus de recherche. Un deuxième objectif est de proposer un framework capable de comparer des voisinages et des sélecteurs dans le contexte d'une procédure de recherche multiopérateur.

2.1 Définitions générales

Cette section a pour but de définir clairement ce que sont un voisinage, la sélection d'un voisin et donc un opérateur, ainsi que les différentes notions associées au processus de recherche.

2.1.1 Voisinage

Soit $\mathcal{S}$ l'espace des solutions candidates. Une relation de voisinage est une relation binaire irréflexive

$\mathcal{N} \subseteq \mathcal{S}^2$ sur l'espace de recherche. Dans la majorité des cas, la relation est aussi symétrique.

2.1.2 Trajectoire de recherche

Soit une relation de voisinage $\mathcal{N}$, nous définissons l'ensemble des trajectoires de recherches par $\mathcal{P}_\mathcal{N} = \{s_1 \cdots s_n \in \mathcal{S}^* | \forall i > 1, (s_{i-1}, s_i) \in \mathcal{N}\}$, où $\mathcal{S}^*$ représente classiquement l'ensemble des mots construits sur $\mathcal{S}$. Donc, toute paire (s, s') d'éléments de $\mathcal{S}$, telles que $(s, s') \in \mathcal{N}^+$, définie une classe d'équivalence sur l'ensemble $\mathcal{P}_\mathcal{N}$, qui correspond à tous les chemins reliant s à s'. Nous notons cet ensemble $\mathcal{P}_\mathcal{N}/(s, s')$. Dans la plupart des cas, le voisinage est complet, c.-à-d. $\forall s, s' \in S, \mathcal{P}_\mathcal{N}/(s, s') \neq \emptyset$.

2.1.3 Distances

En fait, la relation de voisinage définit la structure déclarative de l'espace de recherche. Nous pouvons ainsi définir la distance entre s et s' comme $d_\mathcal{N}(s, s') = min_{p \in \mathcal{P}_\mathcal{N}/(s,s')}|p|$, où $|p|$ est la longueur classique d'un mot. Par définition, nous imposons que $d_\mathcal{N}(s, s) = 0$. Notons qu'il est possible de requérir que $\mathcal{N}$ soit symétrique afin que d soit une distance.

2.1.4 Combinaison de voisinages

Afin d'exprimer des structures de voisinages plus complexes, nous notons $\mathcal{N} \circ \mathcal{N}'$ la composition et $\mathcal{N} \cup \mathcal{N}'$ l'union qui sont les constructeurs de voisinage les plus communément utilisés. Un voisinage composé avec lui-même est noté $\mathcal{N}^2$ et $\mathcal{N}^{n+1} = \mathcal{N} \circ \mathcal{N}^n$.

2.1.5 Paysage de recherche

Considérons maintenant le paysage de recherche, nous introduisons d'abord la relation d'ordre $<$ sur $\mathcal{S}$ correspondant à l'ordre induit par la fonction objectif du problème. Notons que nous considérons ici uniquement les problèmes de minimisation sans perte de généralité.

2.1.6 Paysage opérationnel

Il nous faut maintenant introduire la structure opérationnelle de la recherche locale afin d'explorer la relation de voisinage.

Dans ce contexte, un sélecteur est une fonction qui procède à une sélection sur un voisinage, éventuellement guidé par la relation d'ordre $<$ et défini par $\sigma : \mathcal{S} \times 2^{\mathcal{S}^2} \mapsto \mathcal{S}$ (ici la sélection retourne un unique voisin), tel que $(s, \sigma(s, \mathcal{N})) \in \mathcal{N}^=$ (la clôture réflexive de $\mathcal{S}$ afin d'inclure l'identité). Un opérateur est alors défini par une paire $(\mathcal{N}, \sigma)$.

Considérons à nouveau les chemins induits par un opérateur $\mathcal{P}_o = \bigcup_{n>1}\{s_1 \cdots s_n \in \mathcal{S}^* | o = (\mathcal{N}, \sigma), \forall i > 1, s_i = \sigma(s_{i-1}, \mathcal{N})\}$ Afin de simplifier la notation, nous utilisons $o(s) = \sigma(s, \mathcal{N})$ quand $o = (\mathcal{N}, \sigma)$ puisque o peut être considéré comme une fonction sur S. Nous notons $o \circ o'$ la composition entre opérateurs, o^2 la composition de o avec lui-même et $o^{n+1} = o \circ o^n$.

Notons, ici, que nous avons seulement l'inclusion $\mathcal{P}_o \subseteq \mathcal{P}_\mathcal{N}$, puisque certains chemins du voisinage ne peuvent pas forcément être construits par les opérateurs dès lors qu'ils contiennent un processus de sélection entre les voisins. De plus, même s'il existe un chemin dans $\mathcal{P}_o$ de s à s', il n'existe pas forcément un chemin de s' à s. Du fait de cet aspect non symétrique des opérateurs l'utilisation d'une distance simple sur les chemins créés par les opérateurs n'est pas forcément évidente. Nous pouvons maintenant gérer la recherche locale à voisinages multiples en composant ou en réunissant des relations de voisinage.

2.2 Permutations

Concentrons-nous maintenant sur les permutations qui correspondent à l'encodage que nous utiliserons dans nos problèmes. Notre objectif est de proposer un panorama détaillé des opérateurs possiblement utilisables dans ce contexte.

Soit $\Pi(n)$ l'espace de recherche, c.-à-d. l'ensemble de toutes les permutations de l'ensemble $\{1, \ldots, n\}$. Si $\pi \in \Pi(n)$ et $1 \leq i \leq n$, alors π_i dénote l'élément i dans π.

Comme décrit dans [23] nous pouvons utiliser un ensemble de relations de base de voisinage induites par les permutations de base possibles.

Swap $\mathcal{N}_S$:
$(s, s') \in \mathcal{N}_S$ ssi $s = (\pi_1, \ldots, \pi_i, \pi_{i+1}, \ldots, \pi_n)$ et $s' = (\pi_1, \ldots, \pi_{i+1}, \pi_i, \ldots, \pi_n)$ pour un i.

Échange $\mathcal{N}_E$:
$(s, s') \in \mathcal{N}_E$ ssi $s = (\pi_1, \ldots, \pi_{i-1}, \pi_i, \pi_{i+1}, \ldots, \pi_{j-1}, \pi_j, \pi_{j+1}, \ldots, \pi_n)$ et $s' = (\pi_1, \ldots, \pi_{i-1}, \pi_j, \pi_{i+1}, \ldots, \pi_{j-1}, \pi_i, \pi_{j+1}, \ldots, \pi_n)$ pour un i et un j.

Insertion $\mathcal{N}_I$:
$(s, s') \in \mathcal{N}_I$ ssi $s = (\pi_1, \ldots, \pi_{i-1}, \pi_i, \pi_{i+1}, \ldots, \pi_{j-1}, \pi_j, \pi_{j+1}, \ldots, \pi_n)$ et $s' = (\pi_1, \ldots, \pi_{i-1}, \pi_{i+1}, \ldots, \pi_{j-1}, \pi_i, \pi_j, \pi_{j+1}, \ldots, \pi_n)$ pour un i et un j.

Échange d'arêtes $\mathcal{N}_{EE}$:
$(s, s') \in \mathcal{N}_{EE}$ ssi $s = (\pi_1, \ldots, \pi_{i-1}, \pi_i, \pi_{i+1}, \ldots, \pi_{j-1}, \pi_j, \pi_{j+1}, \ldots, \pi_n)$ et $s' = (\pi_1, \ldots, \pi_i, \pi_j, \pi_{j-1}, \ldots, \pi_{i+1}, \pi_{j+1}, \ldots, \pi_n)$ pour $i + 1 < j$.

$\mathcal{N}^+$ est la clôture transitive de $\mathcal{N}$.

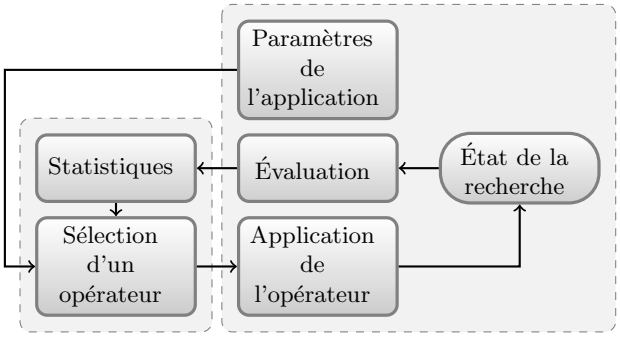


FIGURE 1 – Schéma général

Il apparait clairement que le voisinage construit par $\mathcal{N}_S$ peut également être construit par $\mathcal{N}_E$ et $\mathcal{N}_I$. Des relations d'ordres peuvent donc être définies pour classifier les voisinages afin de mettre l'emphase sur les relations entre les distances qu'ils induisent (voir [23] pour d'avantage de détails).

Nous pouvons maintenant proposer plusieurs fonctions classiques de sélections afin de construire des opérateurs.

Aléatoire : σ_R tel que $\sigma(s, \mathcal{N})$ est n'importe quel s' choisi aléatoirement tel que $(s, s') \in \mathcal{N}$.

Meilleure amélioration : $\sigma_B I$ tel que $\sigma(s, \mathcal{N})$ est un élément minimal s' selon l'ordre $<$, tel que $(s, s') \in \mathcal{N}$.

Meilleure amélioration k : $\sigma_B Ik$ tel que $\sigma(s, \mathcal{N})$ est un élément uniformément sélectionné $s' \in K$, K étant l'ensemble des k-meilleurs éléments selon l'ordre $<$, tel que $(s, s') \in \mathcal{N}$.

Amélioration : σ_I tel que $\sigma(s, \mathcal{N})$ est n'importe quel élément s' tel que $(s, s') \in \mathcal{N}$ et $s' < s$.

Tournois k : $\sigma_T k$ tel que $\sigma(s, \mathcal{N})$ est un élément s' tel que K est un sous-ensemble de k éléments en relation avec s dans $\mathcal{N}$ et s' est le meilleur de ces k éléments.

3 Contrôle d'opérateurs pour la recherche locale

L'objectif de notre méthode est de sélectionner, à partir d'un ensemble donné d'opérateurs, un opérateur approprié à appliquer à chaque itération (Fig. 1).Ceci requiert l'évaluation de l'efficacité des opérateurs basée sur leur comportement antérieur et la sélection d'un opérateur capable de faire progresser le processus de recherche, à des fins soit d'intensification soit de diversification.

Notre objectif est d'avoir une approche aussi générique que possible pour résoudre les problèmes de permutation. À cette fin, notre méthode de résolution se décompose en quatre modules distinct.

– Définition du Problème (DP)
– Gestionnaire de Chemin (GC)
– Gestionnaire d'Opérateurs (GO)
– Conteneur d'Opérateurs (CO)

Du point de vue de l'utilisateur, résoudre un nouveau problème de permutation nécessite seulement de définir une procédure de lecture les données de l'instance et la spécification de la fonction objectif et des contraintes. L'utilisateur peut, s'il le souhaite, ajouter de nouveaux opérateurs au Conteneur d'Opérateurs. Il peut également définir une méthode d'initialisation de la solution initiale s'il ne veut pas utiliser une solution générée aléatoirement.

Le programme est écrit en COMET [26]. Nous estimons que ce langage de programmation destiné à la recherche locale basée sur les contraintes nous offre des perspectives intéressantes puisqu'il facilite la programmation d'algorithmes de recherche locale et qu'il fournit des mécanismes simples de manipulation de voisinages et de sélecteurs. Notre programme se base sur ces fonctionnalités intuitives et peut être considéré comme une extension de COMET. Nous sommes convaincus que la généricité et la simplicité (modulo le minimum de connaissances nécessaire afin de définir le problème) alliées à la facilité intrinsèque d'utilisation de COMET est un pas positif vers la démocratisation de l'utilisation de logiciels d'optimisation.

```
DP (problème de permutation)
Ajouter les opérateurs au CO
Initialiser le GC
Initialiser le GO
s ← solution initiale
s* ← s
répéter
    op ← sélectionner un opérateur
    s ← op(s)
    Mise à jour du GC avec s
    Mise à jour du GO avec les mesures de s
    si s est meilleur que s* alors s* ← s
jusqu'à condition de fin
retourner s*
```

Algorithme 1 : Description algorithmique globale

Notre approche se veut aussi générique que possible mais fait face à certaines limites. En pratique (pour des opérateurs plus complexes que de simples échanges), l'utilisateur doit donner la fonction permettant de calculer le delta de l'évaluation d'une solution candidate puisque qu'elle dépend de la fonction objectif.

3.1 Métriques

Comme mentionné dans l'introduction, un aspect important du contrôle est d'estimer l'équilibre diversification/intensification en utilisant des métriques qui

peuvent évaluer l'efficacité d'un opérateur par rapport à la trajectoire de recherche afin de choisir le prochain opérateur. Nous proposons de traiter simultanément deux critères, la qualité et la diversité, afin de gérer cet équilibre comme un compromis.

3.1.1 Qualité

La qualité est mesurée directement grâce à la fonction objectif. Le changement relatif de qualité en utilisant un opérateur *op* sur une solution s est donné par

$$\Delta Q = \frac{eval(op(s)) - eval(s)}{eval(s) + 1}$$

3.1.2 Distance

La diversité est un concept intuitif lorsque l'on considère des populations de solutions dans le contexte des algorithmes évolutionnaires. Elle l'est moins en recherche locale où une seule solution est produite à chaque itération. Cette notion a été explorée par exemple dans [24] et dans [14]. Nous pourrions considérer la diversité du chemin de la recherche (la séquence de solutions déjà trouvées) ou une fenêtre glissante de celui-ci. À la place nous choisissons d'essayer de mesurer la différence entre le chemin et la solution candidate actuelle $c = op(s)$. Nous proposons deux perspectives différentes : 1) le degré de dissimilitude entre le chemin et c au niveau des variables elles-même ; 2) l'éloignement de c par rapport au chemin en termes de nombres d'opérations les séparant.

La distance L_1 (de Manhattan) entre deux vecteurs p et q est définie par $d_1(p,q) = \sum_{i=1}^{n} |p_i - q_i|$. Nous utilisons une métrique simple mesurant la distance L_1 entre les représentations de la solution candidate et le barycentre du chemin. Les dimensions des points représentant les solutions sont les variables binaires, $x_{a,b}$, la valeur desquelles est 1 dans la solution candidate, où $x_{a,b} = 1$ implique que la variable a à pour valeur b pour les problèmes d'affectation ou que a est suivie de b pour les problèmes d'ordonnancement. Le barycentre correspond donc à la fréquence de $x_{a,b} = 1$ dans le chemin.

Plus formellement, soit $X = \{1, \ldots, n\}$ tel qu'il existe une bijection g de X dans l'ensemble des variables représentant les solutions où Y est le domaine de ces variables. Soit $f_s : X \to Y$ la fonction représentant les valeurs données aux variables de la solution s. Soit $P_{i,j}$ le chemin de l'itération i jusqu'à j, $i \leq j$. Alors

$$d_1^P(c, P_{i,j}) = \frac{1}{n} \times \sum_{k=1}^{n} \left(1 - \frac{occ(P_{i,j}, x_{k,f_c(k)})}{|P_{i,j}|}\right)$$

où $occ(P_{i,j}, x_{a,b})$ correspond au nombre de fois où $x_{a,b} = 1$ dans $P_{i,j}$.

Nous utilisons également la distance de base de voisinage présentée à la section 2, l'idée étant de comparer le chemin de recherche au chemin optimal qui aurait pu être obtenu en utilisant cette relation de voisinage de référence.

Nous définissons ainsi la distance

$$d_{\mathcal{N}}^P(p_k, P_{i,j}) = \sum_{l=i}^{j} \frac{|P_{l,k}|}{d_{\mathcal{N}}(p_l, p_k)} \quad , i \leq j \leq k$$

En utilisant un opérateur suffisamment simple, $d_{\mathcal{N}}^P$ peut être utilisée pour évaluer les caractéristiques exploratoires d'opérateurs plus complexes. À cette fin, nous suggérons l'utilisation de l'opérateur Échange, $\mathcal{N}_E$. Un algorithme permettant de calculer $d_{\mathcal{N}_E}$ est présenté dans [23].

3.2 Sélection d'opérateurs

Nous présentons maintenant le processus de sélection utilisé pour choisir l'opérateur de mouvement à chaque étape. Soient deux vecteurs u et v de même cardinalité p et considérant un problème de maximisation, u *domine* v si $u_k \geq v_k, \forall k \in \{1, \ldots, p\}$ avec au moins une inégalité stricte. Le terme Pareto-dominance est fréquemment employé pour décrire cette relation.

Considérons la population des vecteurs bidimensionnels représentant la performance de chaque opérateur. Dans cet article, la performance correspond à la moyenne ΔQ et d^P calculée pour chaque opérateur sur une fenêtre glissante, spécifique à chaque opérateur, de longueur m. La performance initiale de chaque opérateur est calculée en testant chacun d'eux une fois sur la solution initiale au début de la recherche. Si un opérateur n'a pas été utilisé au cours des m dernières opérations, la fenêtre glissante de cet opérateur ne sera pas vide : elle contiendra au moins un élément (et au plus m éléments) calculés avant ces m itérations.

L'opérateur à utiliser à chaque itération de l'algorithme est sélectionné selon une probabilité proportionnelle à son utilité [20]. Nous définissons l'utilité d'un opérateur comme le nombre d'opérateurs que celui-ci domine augmenté d'un ϵ afin de conserver une utilité non nulle.

4 Expérimentations

Nous testons notre méthode sur le *problème d'affectation quadratique* (QAP) et le *problème du voyageur de commerce asymétrique* (ATSP). Ces problèmes sont choisis car leurs solutions sont facilement représentables par un vecteur de variables. Le QAP modélise le problème qui revient à trouver le coût minimum

d'affectation d'unités de productions à des emplacements, les coûts correspondant à la somme des produits distance-flot [16]. Le ATSP demande de trouver le chemin hamiltonien de poids minimum dans un graphe orienté [7]. La solution initial pour le QAP est générée aléatoirement et l'heuristique de construction « voisin le plus proche » est utilisée pour le ATSP. Dans ces expériences nous utilisons dix opérateurs :

- O1 ($\sigma_I, \mathcal{N}_E$), première amélioration pour l'échange entre deux variables.
- O2 ($\sigma_B I, \mathcal{N}_E$), meilleure amélioration pour l'échange entre deux variables.
- O3 ($\sigma_B I5, \mathcal{N}_E$), choix aléatoire parmi les cinq meilleurs échanges entre deux variables.
- O4 ($\sigma_B I, \mathcal{N}_E)^2$, deux meilleurs échanges consécutifs. Les variables échangées à l'étape une sont interdites à l'étape deux.
- O5 ($\sigma_B I, \mathcal{N}_E)^3$, trois meilleurs échanges consécutifs. Les variables échangées aux étapes précédentes sont interdites à l'étape suivante.
- O6 ($\sigma_T 3!, \mathcal{N}_E^2$), meilleur échange entre 3 variables choisies aléatoirement.
- O7 ($\sigma_T 4!, \mathcal{N}_E^3$), meilleur échange entre 4 variables choisies aléatoirement.
- O8 ($\sigma_T 5!, \mathcal{N}_E^4$), meilleur échange entre 5 variables choisies aléatoirement.
- O9 ($\sigma_T 6!, \mathcal{N}_E^5$), meilleur échange entre 6 variables choisies aléatoirement.
- O10 ($\sigma_R, \mathcal{N}_E^3$), trois échanges aléatoires consécutifs entre deux variables.

Comme décrit plus loin, ces dix opérateurs ne donnent pas de bons résultats pour le ATSP. Nous ajoutons donc l'opérateur O11. Le mouvement 3-opt [7] ($\sigma_B I, \mathcal{N}_{EE}^2$) sélectionne le meilleur voisin obtenu en cassant 3 arêtes et en reconstruisant de nouvelles arêtes de sorte qu'aucun sous-chemin ne soit inversé.

Dans ces expériences, l'accent est mis sur des voisinages construits à partir de $\mathcal{N}_E$, qui semble être un bon voisinage intermédiaire. D'autres voisinages pourraient également être utilisés (d'ailleurs ils peuvent également être exprimés en termes de $\mathcal{N}_E$). Des travaux futurs pourraient étudier des ensembles plus importants de combinaisons comme cela a été fait dans le cadre des algorithmes évolutionnaires dans [19].

4.1 Protocole expérimental

Les instances de test proviennent de QAPLIB [3] pour le QAP et de TSPLIB [21] pour le ATSP. Chaque couple (algorithme, instance) est reproduit 30 fois. La taille de la fenêtre glissante est arbitrairement fixée à 100 et $\epsilon = 1$ pour le processus de sélection. Chaque exécution tourne au maximum pendant 40 000 itérations. Nous utilisons le test non paramétrique des rangs-signés de Wilcoxon [5, 25]. Soit deux algorithmes A et B, l'hypothèse nulle est la suivante : les médianes des distributions des solutions générées par A et B sont équivalentes. L'hypothèse est rejetée avec un seuil de confiance de 95%.

4.2 Résultats et discussion

Le tableau 1 présente les résultats pour le QAP et le tableau 2 ceux du ATSP. Le pourcentage moyen d'écart entre la meilleure valeur connue (BKV) et le choix proportionnel aux valeurs d'utilité suivantes sont donnés : distribution uniforme, qualité, nombre de solutions Pareto-dominées en utilisant la distance d_1^P (ParDom d_1^P) et la distance $d_{\mathcal{N}_E}^P$ (ParDom $d_{\mathcal{N}_E}^P$). Les résultats pour la recherche tabou robuste (RoTS) pour le QAP sont aussi donnés à titre de comparaison (bien entendu des travaux plus récents en recherche locale obtiennent de meilleurs résultats que RoTS, par exemple [12]). Notre objectif ici est simple de donner un marqueur de référence, réécrit en COMET, et de montrer que notre méthode, utilisant un ensemble non optimisé d'opérateurs est capable d'atteindre des résultats intéressants. Les meilleurs résultats pour chaque instance sont indiqués en gras (les résultats de RoTS ne sont pas pris en compte car ils sont meilleurs ou équivalents sur toutes les instances sauf deux). Dans le tableau 2, la colonne ParDom10 contient les résultats lorsque seuls les dix même opérateurs que pour le QAP sont utilisés avec la distance d_1^P.

Pour le QAP ParDom d_1^P et ParDom $d_{\mathcal{N}_E}^P$ partagent la majorité des résultats en gras. Cependant, selon le test de Wilcoxon, ParDom d_1^P semble être le meilleur algorithme lorsqu'il est comparé à la sélection uniforme et à celle basée sur la qualité. A contrario, les résultats de ParDom $d_{\mathcal{N}_E}^P$ ne sont pas statistiquement significatifs. Les résultats semblent donc montrer que ParDom d_1^P est meilleur que ParDom $d_{\mathcal{N}_E}^P$ (l'hypothèse nulle ne peut toutefois pas être rejetée si on les compare l'un à l'autre.

Pour le ATSP, l'utilisation des dix opérateurs utilisés pour le QAP n'est pas concluante. Cela s'explique aisément puisqu'aucun d'entre eux ne prend en compte la nature cyclique des solutions. L'ajout de l'opérateur 3-opt produit une amélioration notable. Avec un unique opérateur spécifique au ATSP, la population des opérateurs demeure très biaisée envers le ATSP. Ceci résulte en une amélioration bien plus visible que dans le cas du QAP lorsque l'on compare les trois méthodes de sélection non triviale avec la sélection uniforme. Ici, ParDom d_1^P et ParDom $d_{\mathcal{N}_E}^P$ partagent les meilleurs résultats avec toutefois la majorité pour ParDom $d_{\mathcal{N}_E}^P$. En termes statistiques, les deux méthodes de sélection par Pareto dominance sur onze opérateurs

TABLE 1 – Résultats expérimentaux pour le QAP.

Instance	BKV	Uniforme	Qualité	ParDom d_1^P	ParDom $d_{\mathcal{N}_E}^P$	RoTS
bur26a	5426670	0.000244	0.001629	**0.000000**	0.004015	0.000000
bur26c	5426795	0.000061	**0.000000**	0.000059	0.000002	0.000000
bur26f	3782044	**0.000000**	**0.000000**	**0.000000**	**0.000000**	0.000000
chr25a	3796	11.790306	10.353003	10.189673	**9.381454**	7.093783
els19	17212548	**0.000000**	**0.000000**	**0.000000**	**0.000000**	0.000000
kra30a	88900	**0.470416**	0.488939	0.499888	0.730034	0.067267
kra30b	91420	0.110698	0.124335	**0.063881**	0.098666	0.023408
nug20	2570	**0.000000**	**0.000000**	**0.000000**	**0.000000**	0.000000
nug30	6124	0.12279556	0.091444	0.057478	**0.050947**	0.014370
sko42	15812	0.163167	0.148832	**0.090817**	0.115608	0.029598
sko49	23386	0.266655	0.194703	**0.186265**	0.193962	0.125203
sko56	34458	0.212781	**0.196955**	0.229497	0.292762	0.118753
tai30a	1818146	1.131385	1.178607	0.794332	**0.633736**	0.512898
tai35a	2422002	1.538266	1.391353	0.943254	**0.745479**	0.762013
tai50a	4941410	1.847374	1.815764	1.377229	**1.363935**	1.391181
tai30b	637117113	0.150888	0.107800	**0.103892**	0.129518	0.026246
tai50b	458821517	**0.173836**	0.186702	0.269760	0.537427	0.150598
wil50	48816	0.076696	**0.074429**	0.079400	0.090216	0.053425

TABLE 2 – Résultats expérimentaux pour le ATSP.

Instances	BKV	ParDom10	Uniforme	Qualité	ParDom d_1^P	ParDom $d_{\mathcal{N}_E}^P$
br17	39	**0.000000**	**0.000000**	**0.000000**	**0.000000**	**0.000000**
p43	5620	0.202847	0.009490	0.002372	0.001779	**0.000593**
ry48p	14422	4.309620	0.661721	0.347155	0.204086	**0.168955**
ft53	6905	12.608255	1.108858	0.517982	0.186338	**0.172822**
ft70	38673	5.891276	0.749791	0.455787	0.080849	**0.048268**
ftv33	1286	7.550544	**0.000000**	**0.000000**	**0.000000**	**0.000000**
ftv35	1473	5.489930	0.495587	0.072415	0.067889	**0.031681**
ftv38	1530	6.141612	0.718954	0.429194	0.305011	**0.259259**
ftv44	1613	9.070056	0.725356	0.378177	**0.237652**	0.252118
ftv47	1776	11.006006	0.478604	0.191441	0.138889	**0.114489**
ftv55	1608	14.195688	0.972222	0.213516	0.136816	**0.093284**
ftv64	1839	17.130687	1.386623	0.781222	**0.554649**	0.580025
ftv70	1950	17.042735	1.540171	0.919658	**0.635897**	0.637607
ftv90	1579	25.429597	2.180705	1.253958	0.975301	**0.821195**
ftv100	1788	24.571216	2.839299	1.498881	1.168904	**1.047726**
ftv110	1958	31.089547	4.375213	2.667688	2.378277	**2.311883**
ftv120	2166	25.386273	3.464143	2.368421	2.136042	**2.132964**
ftv130	2307	22.831961	4.838896	2.781390	2.417281	**2.265569**
ftv140	2420	32.836088	4.720386	3.286501	3.004132	**2.965565**
ftv150	2611	32.370739	5.581514	3.993361	**3.150772**	3.292481
ftv160	2683	35.242887	5.998261	3.467511	3.473723	**3.334576**
ftv170	2755	33.393829	5.929825	3.680581	**3.097816**	3.553539
kro124p	36230	18.522587	2.327813	1.358451	1.150520	**1.055479**
rbg323	1326	7.986425	0.072901	0.012569	**0.000000**	**0.000000**
rbg358	1163	9.203210	0.005732	**0.000000**	**0.000000**	**0.000000**
rbg403	2465	1.150778	**0.000000**	**0.000000**	**0.000000**	**0.000000**
rbg443	2720	1.455882	**0.000000**	**0.000000**	**0.000000**	**0.000000**

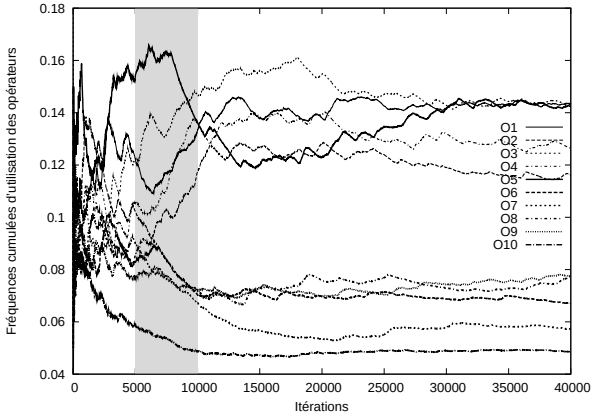
sont meilleures que la sélection uniforme ainsi que celle basée sur la qualité. Lorsque l'on compare ParDom d_1^P à ParDom $d_{\mathcal{N}_E}^P$, l'hypothèse nulle peut être rejetée. Cela démontre clairement que ParDom $d_{\mathcal{N}_E}^P$ est la mieux adaptée au ATSP.

La figure 2(a) représente la fréquences cumulées de l'application des opérateurs pour une exécution arbitraire du QAP et la figure 3 représente la même chose pour le ATSP. On peut observer que les opérateurs sont clairement séparés en deux groupes : un dont la fréquence est supérieure à la moyenne (0,1) et l'autre inférieure. Le détail nous révèle que le premier est le groupe qui améliore le plus la qualité alors que le second est celui qui perturbe le plus les solutions sans apporter de changement positif à la qualité.

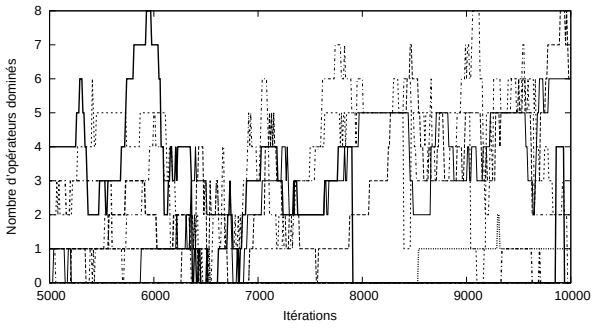
L'opérateur qui est sélectionné le plus souvent est celui qui démontre sa constance dans l'amélioration de la solution tout en modifiant un certain nombre de variables au cours de ses cent dernières applications. Comme on peut le voir à la figure 2(a), l'opérateur O5 se comporte très bien au début de la recherche. Sa performance baisse ensuite pour augmenter à nouveau jusqu'au niveau des autres « bons » opérateurs, la recherche stagnant à la fin. Ceci illustre le fait qu'un opérateur n'est pas toujours le meilleur tout au long de la recherche et qu'il est important pour le mécanisme de sélection d'être influencé par l'étape à laquelle se trouve la recherche. A contrario, dans la figure 3, l'opérateur O11 est constamment l'opérateur le plus souvent sélectionné. C'est un résultat auquel on pouvait s'attendre puisque c'est le seul opérateur spécifique au ATSP. Il se comporte donc forcément mieux que les autres opérateurs. Les opérateurs qui améliorent la qualité pour le QAP gardent toutefois un intérêt pour le ATSP.

Afin d'expliquer pourquoi les opérateurs améliorant la qualité sont sélectionnés le plus souvent, on peut remarquer que la modification d'une solution pour la rendre meilleure requiert également la modification de ses variables et donc de la distance depuis la dernière solution. Les opérateurs améliorant la qualité sont donc plus enclin à dominer les opérateurs dont la seule action est de causer des perturbations dans la solution.

La figure 2(b) montre le nombre d'opérateurs dominés par chaque opérateur sur une partie de la recherche de la figure 2(a) (région grisée). On peut observer qu'à certains moments de la recherche un opérateur domine presque tous les autres, ayant ainsi la probabilité la plus grande d'être sélectionné. Cette probabilité accrue se reflète dans les courbes de fréquences cumulées. On peut observer que pendant les quelques centaines d'itérations avant l'itération 7 000, aucun opérateur n'est considéré comme bien bien supérieur aux



(a) Fréquences cumulées de sélection.



(b) Nombre d'opérateurs dominés correspondant aux itérations grisées en (a).

FIGURE 2 – Une exécution de l'instance tai50a (QAP).

autres. Ceci transforme le processus de sélection en une simple sélection uniforme. Une sélection plus discriminante émerge ensuite, des opérateurs différents étant apparemment mieux adaptés à la suite de la recherche.

Une autre observation intéressante, plus particulièrement à la figure 3, est que l'augmentation significative de la fréquence de sélection d'un opérateur implique souvent l'inverse pour un autre opérateur.

La figure 4 est un instantané d'une partie de la recherche sur une instance de QAP. Elle représente la meilleure valeur obtenue, la valeur courante de la fonction objectif et la distance du chemin à la nouvelle solution. Elle met en valeur que le contrôle, gérant le compromis entre qualité et diversité, est capable d'échapper à des minima locaux et également d'atteindre de bonnes solutions. La corrélation entre notre mesure de distance et la qualité apparait aussi clairement.

Bien que cet article ne contienne pas de résultats détaillés, nous pouvons noter que si cinq clones d'un opérateur ne faisant rien sont rajoutés, l'écart de per-

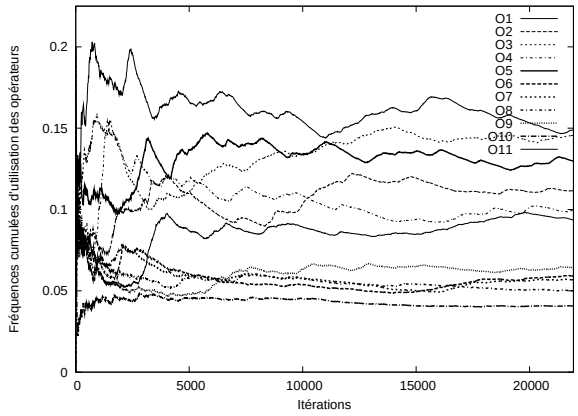


FIGURE 3 – Une exécution de l'instance ftv55 (ATSP).

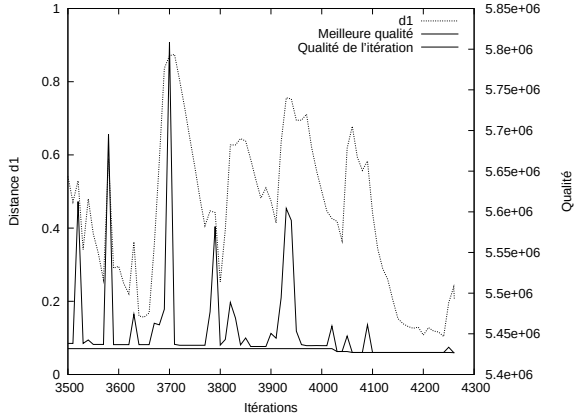


FIGURE 4 – Partie finale de la recherche pour une exécution de l'instance bur26a (QAP).

formance entre les méthodes proposées de sélection et la sélection uniforme s'agrandit. Si l'on utilise que le meilleur opérateur pour l'intensification et le meilleur opérateur pour la diversification avec ϵ égale à 0,1 (1 étant trop proche de la sélection uniforme), les résultats sont moins bons que pour dix opérateurs.

5 Conclusion

Dans cet article nous avons présenté une méthode générique de gestion d'opérateurs de recherche locale pour les problèmes de permutation ainsi que deux métriques pour mesurer la distance entre une solution et une partie du chemin de recherche. Les résultats des tests sur le QAP et le ATSP montrent que l'algorithme fonctionne mais qu'il peut être grandement amélioré.

Le processus de sélection favorise les opérateurs qui maintiennent un compromis coopératif entre l'intensification et la diversification, c.-à-d. les opérateurs au milieu du front Pareto. La prochaine étape de notre travail sera d'inclure un élément réactif au processus. Une autre question importante est la gestion des redémarrages (et pas seulement de petites perturbations), soit comme une redémarrage « externe » dépendant d'une certaine condition soit comme un restart « interne », c.-à-d. un opérateur en lui-même. Il va de soi que l'évaluation de la généricité de notre approche sur d'autres problèmes est requise, et plus particulièrement des problèmes avec des contraintes et des problèmes qui requièrent l'exploration de solutions non réalisables.

Remerciements. Ces travaux sont financés par Microsoft Research à travers le PhD Scholarship Programme.

Références

[1] Roberto Battiti, Mauro Brunato, and Franco Mascia. *Reactive Search and Intelligent Optimization*. Springer Publishing Company, Incorporated, 2008.

[2] Roberto Battiti and Giampietro Tecchiolli. The Reactive Tabu Search. *INFORMS JOURNAL ON COMPUTING*, 6(2) :126–140, January 1994.

[3] Rainer E. Burkard, Stefan E. Karisch, and Franz Rendl. QAPLIB – A Quadratic Assignment Problem Library. *Journal of Global Optimization*, 10(4) :391–403, June 1997.

[4] Edmund Burke, Graham Kendall, Jim Newall, Emma Hart, Peter Ross, and Sonia Schulenburg. Hyper-Heuristics : An Emerging Direction in Modern Search Technology. In Fred Glover and Gary Kochenberger, editors, *Handbook of Metaheuristics*, volume 57, pages 457–474. Springer New York, 2003.

[5] Marco Chiarandini, Luís Paquete, Mike Preuss, and Enda Ridge. Experiments on metaheuristics : Methodological overview and open issues. Technical Report DMF-2007-03-003, The Danish Mathematical Society, 2007.

[6] Luis DaCosta, Álvaro Fialho, Marc Schoenauer, and Michèle Sebag. Adaptive operator selection with dynamic multi-armed bandits. In *Proceedings of the 10th annual conference on Genetic and evolutionary computation*, pages 913–920, Atlanta, GA, USA, 2008. ACM.

[7] Gregory Gutin and Abraham P. Punnen. *The traveling salesman problem and its variations*. Springer, 2002.

[8] Y. Hamadi, E. Monfroy, and F. Saubion. *Hybrid Optimization : The Ten Years of CPAIOR*, chapter What Is Autonomous Search ? Springer, 2010.

[9] Pierre Hansen and Nenad Mladenović. Variable Neighborhood Search. In Fred Glover and Gary Kochenberger, editors, *Handbook of Metaheuristics*, volume 57, pages 145–184. Springer New York, 2003.

[10] Holger Hoos and Thomas Stützle. *Stochastic Local Search : Foundations & Applications*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004.

[11] Bin Hu and Günther R. Raidl. Variable Neighborhood Descent with Self-Adaptive Neighborhood-Ordering. In *Proc. of the 7th EU Meeting on Adaptive, Self-Adaptive and Multilevel Metaheuristics*, 2006.

[12] Mohamed Saifullah Hussin and Thomas Stützle. Hierarchical Iterated Local Search for the Quadratic Assignment Problem. In *Hybrid Metaheuristics, 6th International Workshop, HM 2009,*, volume 5818 of *Lecture Notes in Computer Science*, pages 115–129. Springer, 2009.

[13] Lester Ingber. Adaptive simulated annealing (ASA) : Lessons learned. *Control and Cybernetics*, 25, 1996. Control and Cybernetics 25 (1996) 33-54.

[14] 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).

[15] Alexandre Linhares and Horacio Hideki Yanasse. Search intensity versus search diversity : a false trade off ? *Applied Intelligence*, 32(3) :279–291, 2010.

[16] Eliane Maria Loiola, Nair Maria Maia de Abreu, Paulo Oswaldo Boaventura-Netto, Peter Hahn, and Tania Querido. A survey for the quadratic assignment problem. *European Journal of Operational Research*, 176(2) :657–690, January 2007.

[17] Helena Lourenço, Olivier Martin, and Thomas Stützle. Iterated Local Search. In Fred Glover and Gary Kochenberger, editors, *Handbook of Metaheuristics*, volume 57, pages 320–353. Springer New York, 2003.

[18] J. Maturana and F. Saubion. A compass to guide genetic algorithms. In G. Rudolph et al., editor, *Parallel Problem Solving from Nature - PPSN X, 10th International Conference Dortmund, Germany, September 13-17, 2008, Proceedings*, volume 5199 of *LNCS*, pages 256–265. Springer, 2008.

[19] Jorge Maturana, Frédéric Lardeux, and Frédéric Saubion. Autonomous operator management for evolutionary algorithms. *Journal of Heuristics*, 2010.

[20] Alexander Nareyek. Choosing search heuristics by non-stationary reinforcement learning. In *Metaheuristics : computer decision-making*, pages 523–544. Kluwer Academic Publishers, 2004.

[21] Gerhard Reinelt. TSPLIB – A Traveling Salesman Problem Library. *INFORMS JOURNAL ON COMPUTING*, 3(4) :376–384, January 1991.

[22] Julien Robet, Frédéric Lardeux, and Frédéric Saubion. Autonomous Control Approach for Local Search. In *Engineering Stochastic Local Search Algorithms. Designing, Implementing and Analyzing Effective Heuristics*, pages 130–134. Springer-Verlag, Berlin, Heidelberg, 2009.

[23] Tommaso Schiavinotto and Thomas Stützle. A review of metrics on permutations for search landscape analysis. *Comput. Oper. Res.*, 34(10) :3143–3153, 2007.

[24] Alain Sidaner, Olivier Bailleux, and Jean-Jacques Chabrier. Measuring the Spatial Dispersion of Evolutionary Search Processes : Application to Walksat. In *Artificial Evolution, 5th International Conference, Evolution Artificielle, EA 2001*, volume 2310 of *LNCS*, pages 77–90. Springer, 2001.

[25] Peter Sprent. *Applied Nonparametric Statistical Methods*. Chapman & Hall, London, 1989.

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

[27] Nadarajen Veerapen and Frédéric Saubion. Pareto Autonomous Local Search. In Xin Yao and Carlos A. Coelho Coelho, editors, *Proc. Fifth International Conference on Learning and Intelligent Optimization (LION5), Rome, Jan 17-21, 2011*, Springer, Lecture Notes on Computer Science, to appear.

Programmation par contraintes pour la synthèse de contrôleur

Gérard Verfaillie **Cédric Pralet**

Onera - The French Aerospace Lab, F-31055, Toulouse, France
{Gerard.Verfaillie,Cedric.Pralet}@onera.fr

Résumé

Dans ce papier, nous montrons comment le problème de synthèse d'un contrôleur d'un système dynamique satisfaisant certaines propriétés de sûreté, en contexte éventuellement non déterministe et partiellement observable, peut être modélisé comme un pur problème de satisfaction de contraintes, en remplaçant la propriété d'atteignabilité par une propriété dite d'atteignabilité faible. Nous montrons, d'abord sur un exemple jouet illustratif, puis sur un exemple réel de contrôle d'un sous-système d'un satellite, comment des outils standards de programmation par contraintes peuvent être utilisés pour modéliser et résoudre le problème de synthèse de contrôleur. Nous concluons sur les forces et les faiblesses de l'approche proposée.

Abstract

In this paper, we show how the problem of synthesis of a controller for a dynamic system that must satisfy some safety properties, possibly in a non deterministic and partially observable setting, can be modeled as a pure constraint satisfaction problem, by replacing the reachability property by a so-called weak reachability property. We show, first on a toy illustrative example, then on a real-world example of control of a satellite subsystem, how standard constraint programming tools can be used to model and solve the controller synthesis problem. We conclude with the strengths and weaknesses of the proposed approach.

1 Le problème de synthèse de contrôleur

1.1 Une vue informelle

Dans ce papier, nous nous intéressons au contrôle en boucle fermée de systèmes dynamiques (voir la figure 1). Plus précisément, nous nous intéressons à des contrôleurs logiciels implémentés sur des calculateurs digitaux (l'essentiel des contrôleurs modernes), qui n'agissent pas de façon continue sur le système qu'ils contrôlent, mais de façon discrète, par étapes successives.

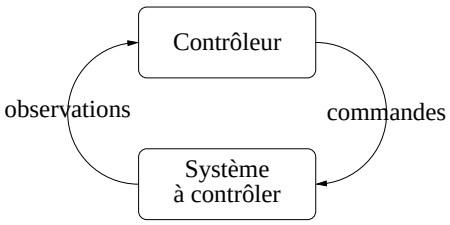


FIGURE 1 – Contrôle en boucle fermée d'un système dynamique

A chaque étape, le contrôleur collecte des informations sur le système à contrôler (observations) et prend de façon réactive une décision de contrôle (commandes) en fonction des informations recueillies. Les observations peuvent venir du système lui-même, d'autres systèmes avec lesquels il est en interaction ou d'opérateurs humains. Elles peuvent contenir des informations synthétiques sur les observations et les commandes passées (mémoire du contrôleur). En sens inverse, les commandes sont adressées au système lui-même ou à d'autres systèmes ou sont des informations à destination des opérateurs.

Nous supposons que l'évolution du système sous l'impact des commandes (transitions) est non déterministe et markovienne : plusieurs transitions sont possibles à partir de l'état et de la commande courants ; la transition effective ne dépend que de l'état et de la commande courants ; elle ne dépend pas des états et commandes précédents. Mais nous ne supposons pas la connaissance d'une distribution de probabilité sur l'ensemble des transitions possibles : seules sont distinguées les transitions possibles et impossibles. De plus, nous ne supposons pas que l'état réel du sys-

tème à chaque étape soit connu du contrôleur : seules des observations sont disponibles. Nous faisons par contre l'hypothèse que l'ensemble des états possibles du système, l'ensemble des observations possibles et l'ensemble des commandes possibles sont discrets et finis.

Nous nous intéressons à des propriétés dites de sûreté, c'est-à-dire à des propriétés qui doivent être satisfaites par toute transition parcourue par le système. Dans un tel contexte, le problème de synthèse de contrôleur consiste à construire hors-ligne (avant la mise en service du contrôleur) ce qu'on appelle une politique, c'est-à-dire une fonction qui associe à chaque ensemble d'observations un ensemble de commandes et qui, en dépit du non déterminisme des transitions, garantisse que toute transition parcourue par le système satisfasse les propriétés de sûreté.

La section 1 introduit le problème de synthèse de contrôleur. La section 2 présente un modèle CSP de ce problème, utilisant la notion d'atteignabilité faible. La section 3 montre comment l'approche proposée peut être appliquée au problème de contrôle d'un sous-système d'un satellite. La section 4 conclut sur les forces et les faiblesses de cette approche.

1.2 Une définition formelle

Le problème de synthèse de contrôleur peut être formellement défini de la façon suivante.

Données Les données du problème sont :

- une séquence finie S de variables d'état du système ;
- une sous-séquence $O \subseteq S$ de variables d'état observables ;
- une séquence finie C de variables de commande ;
- un ensemble $I \subseteq \mathbf{d}(S)$ d'états initiaux possibles du système, supposé non vide ;
- un ensemble $T \subseteq \mathbf{d}(S) \times \mathbf{d}(C) \times \mathbf{d}(S)$ de transitions possibles ;
- un ensemble $P \subseteq \mathbf{d}(S) \times \mathbf{d}(C) \times \mathbf{d}(S)$ de transitions acceptables.

Toute variable x d'état ou de commande est supposée avoir un domaine de valeurs fini, noté $\mathbf{d}(x)$. Si X est une séquence de variables, le produit cartésien des domaines de valeurs des variables de X est noté $\mathbf{d}(X)$. Si X est une séquence de variables et Y une sous-séquence et si A est une affectation de X, $A_{\downarrow Y}$ désigne l'affectation de Y (projection).

Les ensembles I, T et P peuvent être implicitement représentés par des ensembles (finis) de contraintes.

Politique Une politique π est une fonction partielle de $\mathbf{d}(O)$ dans $\mathbf{d}(C)$, partielle parce qu'une politique ne doit pas être définie pour toutes les observations possibles, mais uniquement pour celles qui sont atteignables (associées à au moins un état atteignable). Soit $df_\pi \subseteq \mathbf{d}(O)$ le domaine de définition de π.

États atteignables Étant donnée une politique π, l'ensemble r_π des états qui sont atteignables à partir de l'état initial en suivant π (r pour *reachability*) peut être défini de la façon suivante. Si $r_{\pi,k}$ est l'ensemble des états atteignables en au plus de k étapes à partir d'un état initial, on a :

$$\forall s \in \mathbf{d}(S),\ r_{\pi,0}(s) = I(s) \tag{1}$$

$$\begin{aligned} &\forall k, 1 \leq k \leq |\mathbf{d}(S)| - 1, \\ &r_{\pi,k}(s) = r_{\pi,k-1}(s) \vee (\exists s' \in \mathbf{d}(S), \\ &r_{\pi,k-1}(s') \wedge df_\pi(s'_{\downarrow O}) \wedge T(s', \pi(s'_{\downarrow O}), s)) \end{aligned} \tag{2}$$

$$r_\pi(s) = \max_{0 \leq k \leq |\mathbf{d}(S)|-1} r_{\pi,k}(s) \tag{3}$$

L'équation 1 indique qu'un état est atteignable en 0 étape si et seulement si il est un état initial possible. L'équation 2 indique qu'un état est atteignable en k étapes si et seulement si il est atteignable en $k-1$ étapes ou si une transition est possible depuis un état atteignable en $k-1$ étapes. Enfin, l'équation 3 indique qu'un état est atteignable si et seulement si il est atteignable en k étapes, avec $0 \leq k \leq |\mathbf{d}(S)| - 1$. En effet, au pire, tous les états sont atteignables, il existe un seul état initial possible et un seul nouvel état est atteignable chaque fois que k est incrémenté de 1. Il faut souligner que, suivant cette définition, l'ensemble des états atteignables dépend de la politique retenue.

Exigences sur la politique Les exigences sur la politique sont les suivantes :

$$\forall s \in \mathbf{d}(S), \quad r_\pi(s) \rightarrow df_\pi(s_{\downarrow O}) \tag{4}$$

$$\begin{aligned} &r_\pi(s) \rightarrow (\exists s' \in \mathbf{d}(S), \\ &T(s, \pi(s_{\downarrow O}), s')) \end{aligned} \tag{5}$$

$$\begin{aligned} &r_\pi(s) \rightarrow (\forall s' \in \mathbf{d}(S), \\ &T(s, \pi(s_{\downarrow O}), s') \rightarrow P(s, \pi(s_{\downarrow O}), s')) \end{aligned} \tag{6}$$

L'équation 4 spécifie que la politique doit être définie pour tout état atteignable. L'équation 5 spécifie que la politique ne doit pas introduire de blocage : pour tout état atteignable, en suivant la politique, il existe une transition possible. Enfin, l'équation 6 spécifie que la

politique doit être "acceptable" : pour tout état atteignable, en suivant la politique, toute transition possible est acceptable. Il faut souligner que ces exigences ne doivent pas être satisfaites sur tous les états possibles, mais seulement sur ceux qui sont atteignables à partir de l'état initial en suivant la politique retenue.

1.3 Un exemple jouet

À titre d'illustration, considérons l'exemple jouet utilisé dans [17]. On considère un robot capable de se mouvoir sur la grille de la figure 2 où les murs apparaissent en gras. Initialement, le robot se trouve sur l'une des deux places d'abscisse $x = 2$. À chaque étape, le robot se trouve sur une place p qu'il ne connaît pas directement. Il observe seulement les murs présents immédiatement autour de p. À chaque étape, il se déplace vers le nord, le sud, l'est ou l'ouest. Il n'est pas autorisé à rester sur place. En raison de la présence des murs, seuls certains de ces mouvements sont réalisables. Le robot doit éviter la place marquée X $(x = 3, y = 1)$ considérée comme dangereuse.

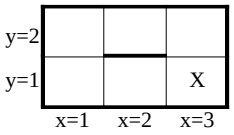


FIGURE 2 – Problème de contrôle d'un robot sur une grille

Pour modéliser ce problème, on considère six variables d'état : $S = \{x, y, w_N, w_S, w_E, w_W\}$. x et y représentent la position du robot, avec $\mathbf{d}(x) = [1..3]$ et $\mathbf{d}(y) = [1..2]$. w_N, w_S, w_E et w_W sont des variables booléennes qui représentent la présence (ou non) d'un mur au nord, au sud, à l'est et à l'ouest de la place courante. Seules les variables d'état w_N, w_S, w_E et w_W sont observables par le robot : $O = \{w_N, w_S, w_E, w_W\}$. On considère une seule variable de commande m qui représente le mouvement du robot, avec $\mathbf{d}(m) = \{m_N, m_S, m_E, m_W\}$ (quatre mouvements possibles).

L'ensemble I des états initiaux possibles est défini par les cinq contraintes unaires suivantes : $x = 2$, w_N, w_S, $\neg w_E$ et $\neg w_W$.

L'ensemble T des transitions possibles est défini par les contraintes suivantes :

$$w_N \rightarrow (m \neq m_N)$$
$$w_S \rightarrow (m \neq m_S)$$
$$w_E \rightarrow (m \neq m_E)$$
$$w_W \rightarrow (m \neq m_W)$$

$$x' = x + (m = m_E) - (m = m_W)$$
$$y' = y + (m = m_N) - (m = m_S)$$

$$w'_N = (y' = 2 \vee (y' = 1 \wedge x' = 2))$$
$$w'_S = (y' = 1 \vee (y' = 2 \wedge x' = 2))$$
$$w'_E = (x' = 3)$$
$$w'_W = (x' = 1)$$

Les quatre premières contraintes traduisent les mouvements irréalisables du fait de la présence des murs. Les deux suivantes expriment les transitions possibles qui, dans ce cas, sont déterministes : une seule transition possible associée à chaque paire état-commande. Pour toute variable d'état z, z' représente sa valeur à l'étape suivante. Les quatre dernières contraintes traduisent la connaissance disponible sur la topologie de la grille.

L'ensemble P des transitions acceptables est défini par la contrainte binaire $\neg((x' = 3) \wedge (y' = 1))$.

1.4 Méthodes existantes

La méthode de loin la plus utilisée pour construire un contrôleur consiste à le spécifier en utilisant un langage de programmation généraliste ou spécialisé, par exemple un langage de la famille des langages synchrones [10]. Une fois le contrôleur programmé, ses propriétés peuvent être vérifiées, soit expérimentalement par simulation, soit formellement en utilisant des outils de preuve ou de *model-checking* [5].

La synthèse de contrôleur est une approche alternative, qui vise la construction automatique d'un contrôleur à partir des propriétés physiques du système à contrôler et des exigences sur le système contrôlé. Le contrôleur produit est valide par construction et aucune vérification n'est théoriquement à réaliser après construction.

Les premiers travaux en synthèse de contrôleur utilisent les automates et la théorie des langages [19]. Par la suite, de nombreux travaux utilisent les automates pour représenter le système physique et la logique temporelle pour exprimer les exigences [15] (voir par exemple les outils ANZU et RATSY [12, 20] qui supposent tous deux une observabilité complète). Certains travaux font appel au μ-calcul [1]. L'outil MBP (*Model-Based Planner* [14]) utilise des techniques dites

de *symbolic model-checking*, à base de BDD (*Binary Decision Diagrams*), pour synthétiser des contrôleurs qui garantissent qu'un but est atteint en dépit du non déterminisme et de l'observabilité partielle [3]. Des algorithmes de recherche génériques pour la synthèse de contrôleurs à mémoire finie sont proposés dans [4, 17], avec les mêmes hypothèses de non déterminisme et d'observabilité partielle (avec cependant certaines restrictions dans [4]).

On doit souligner la proximité entre le problème de synthèse de contrôleur et les (PO)MDP (*(Partially Observable) Markov Decision Processes* [18]) pour lesquels des algorithmes à base de programmation dynamique sont les plus utilisés. La première différence entre les deux problèmes est que, dans les (PO)MDP, une distribution de probabilité conditionnelle sur les états résultant d'une transition et sur les observations résultant d'un état est supposée être disponible. La seconde différence est que les exigences prennent dans les (PO)MDP la forme d'un critère additif à optimiser.

On doit aussi souligner la différence entre le problème de synthèse de contrôleur et le problème de planification en Intelligence Artificielle [8]. En planification, on s'intéresse à des propriétés d'atteignabilité : un état but doit être atteint et le contrôle est supposé être stoppé une fois le but atteint. En synthèse de contrôleur, on s'intéresse à des propriétés de sûreté qui doivent être satisfaites tout au long de la trajectoire du système et le contrôle est supposé sans fin.

2 Formulation comme un problème de satisfaction de contraintes

2.1 Difficultés

Supposons que l'on associe à chaque observation $o \in \mathbf{d}(O)$ une variable $\pi(o)$ de domaine $\mathbf{d}(C) \cup \{\bot\}$, représentant la commande à appliquer lorsque o est observée. La valeur $\bot$ sert à représenter l'absence de commande. Supposons que l'on associe à chaque état $s \in \mathbf{d}(S)$, les variables suivantes :

- une variable booléenne $r_\pi(s)$ qui représente le fait que s soit atteignable (ou non) ;
- pour tout $k, 0 \leq k \leq |\mathbf{d}(S)| - 1$, une variable booléenne $r_{\pi,k}(s)$ qui représente le fait que s soit atteignable (ou non) en au plus k étapes.

Si l'on remplace $df_\pi(o)$ par $\pi(o) \neq \bot$, les équations 1 à 6 forment un problème de satisfaction de contraintes P (CSP [21]) qui modélise précisément le problème de synthèse de contrôleur. Malheureusement, le nombre de variables de P est prohibitif, essentiellement du fait des variables $r_{\pi,k}(s)$: pour chaque état $s \in \mathbf{d}(S)$, on a $|\mathbf{d}(S)|$ variables $r_{\pi,k}(s)$, d'où au total $|\mathbf{d}(S)|^2$ variables $r_{\pi,k}(s)$. Pour contourner cette difficulté, nous allons utiliser une relaxation de la propriété d'atteignabilité que nous appellons atteignabilité faible.

2.2 Atteignabilité et atteignabilité faible

Nous adoptons la définition suivante de l'atteignabilité faible : une relation wr_π est une relation d'atteignabilité faible associée à une politique π (wr pour *weak reachability*) si et seulement si elle satisfait les deux équations suivantes :

$$\forall s \in \mathbf{d}(S), \quad I(s) \rightarrow wr_\pi(s) \tag{7}$$

$$\forall s, s' \in \mathbf{d}(S), \quad (wr_\pi(s) \wedge df_\pi(s_{\downarrow O}) \wedge T(s, \pi(s_{\downarrow O}), s')) \rightarrow wr_\pi(s') \tag{8}$$

L'équation 7 exprime que, si un état est un état initial possible, il est faiblement atteignable. L'équation 8 exprime que, si un état s est faiblement atteignable et si une transition est possible depuis l'état s vers un autre état s', l'état s' est lui aussi faiblement atteignable. À partir de cette définition de l'atteignabilité faible, les quatre propriétés suivantes peuvent être établies.

Propriété 1 *Soit π une politique. La relation d'atteignabilité associée r_π est unique. Il peut par contre exister plusieurs relations d'atteignabilité faible associées wr_π.*

La relation d'atteignabilité est unique parce qu'elle est définie par les équations 1 à 3 qui sont toutes des égalités. L'existence possible de plusieurs relations d'atteignabilité faible est mise en évidence par l'exemple de la figure 3 où est représenté le graphe d'atteignabilité associée à une politique. Dans ce graphe, les nœuds représentent les états et les arcs des transitions possibles.

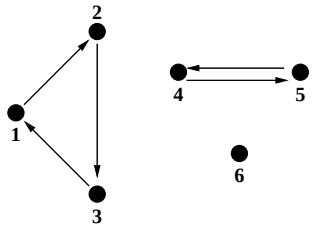


FIGURE 3 – Exemple de graphe d'atteignabilité associée à une politique

Sur cet exemple, si 1 est le seul état initial possible, la relation d'atteignabilité r est définie par l'ensemble d'états $\{1, 2, 3\}$. Par contre, les quatre relations définies respectivement par les ensembles d'états

$\{1,2,3\}$, $\{1,2,3,4,5\}$, $\{1,2,3,6\}$ et $\{1,2,3,4,5,6\}$ satisfont toutes les équations 7 et 8 et sont donc toutes des relations d'atteignabilité faible.

Propriété 2 *Soient π une politique, r_π la relation d'atteignabilité associée et wr_π une relation d'atteignabilité faible associée. r_π est un sous-ensemble de wr_π.*

Pour établir cette propriété, montrons que $\forall s' \in \mathbf{d}(S), r_\pi(s') \rightarrow wr_\pi(s')$.

Si $r_\pi(s')$, il existe k, $0 \leq k \leq |\mathbf{d}(S)| - 1$, tel que $r_{\pi,k}(s')$ (d'après l'équation 3). Montrons par récurrence sur k que $\forall k \geq 0, \forall s' \in \mathbf{d}(S), r_{\pi,k}(s') \rightarrow wr_\pi(s')$.

Pour $k = 0$, si $r_{\pi,0}(s')$, on a $I(s')$ (d'après l'équation 1) et donc $wr_\pi(s')$ (d'après l'équation 7).

Pour $k > 0$, supposons que $\forall s' \in \mathbf{d}(S), r_{\pi,k-1}(s') \rightarrow wr_\pi(s')$. Si $r_{\pi,k}(s')$, on a (d'après l'équation 2) soit $r_{\pi,k-1}(s')$ et donc $wr_\pi(s')$ (d'après l'hypothèse de récurrence), soit $\exists s \in \mathbf{d}(S)\,,\, r_{\pi,k-1}(s) \wedge df_\pi(s_{\downarrow O}) \wedge T(s, \pi(s_{\downarrow O}), s')$ et donc $\exists s \in \mathbf{d}(S)\,,\, wr_\pi(s) \wedge df_\pi(s_{\downarrow O}) \wedge T(s, \pi(s_{\downarrow O}), s')$ (toujours d'après l'hypothèse de récurrence) et finalement $wr_\pi(s')$ (d'après l'équation 8). On en déduit que $\forall s' \in \mathbf{d}(S), r_{\pi,k}(s') \rightarrow wr_\pi(s')$.

En conséquence, $\forall s' \in \mathbf{d}(S), r_\pi(s') \rightarrow wr_\pi(s')$.

Propriété 3 *Soit π une politique. La relation d'atteignabilité associée r_π est une relation d'atteignabilité faible associée à π.*

Pour l'établir, il suffit de montrer que r_π satisfait les équations 7 et 8 qui définissent l'atteignabilité faible. Elle les satisfait du fait de la définition de l'atteignabilité (équations 1 à 3).

Propriété 4 *Soit π une politique. La relation d'atteignabilité associée r_π est l'unique plus petite relation d'atteignabilité faible associée à π (plus petite au sens de l'inclusion et donc, dans ce cas, de la cardinalité).*

Cette propriété est la conséquence immédiate des propriétés 1, 2 et 3.

2.3 Formulation proposée

En utilisant la notion d'atteignabilité faible, nous proposons la formulation suivante du problème de synthèse de contrôleur.

Variables Cette formulation utilise deux ensembles de variables :

- pour chaque observation $o \in \mathbf{d}(O)$, une variable $\pi(o)$ de domaine $\mathbf{d}(C) \cup \{\perp\}$, représentant la commande à appliquer lorsque o est observée ;
- pour chaque état $s \in \mathbf{d}(S)$, une variable booléenne $wr_\pi(s)$ qui représente le fait que s soit faiblement atteignable ou non.

Contraintes Les contraintes à satisfaire sont définies par les équations suivantes :

$$\forall s \in \mathbf{d}(S), \quad I(s) \rightarrow wr_\pi(s) \tag{9}$$

$$\forall s, s' \in \mathbf{d}(S), \quad (wr_\pi(s) \wedge T(s, \pi(s_{\downarrow O}), s')) \rightarrow wr_\pi(s') \tag{10}$$

$$\forall s \in \mathbf{d}(S), \quad wr_\pi(s) \rightarrow (\pi(s_{\downarrow O}) \neq \perp) \tag{11}$$

$$wr_\pi(s) \rightarrow (\exists s' \in \mathbf{d}(S), T(s, \pi(s_{\downarrow O}), s')) \tag{12}$$

$$wr_\pi(s) \rightarrow (\forall s' \in \mathbf{d}(S), T(s, \pi(s_{\downarrow O}), s') \rightarrow P(s, \pi(s_{\downarrow O}), s')) \tag{13}$$

Les équations 9 et 10 sont de simples copies des équations 7 et 8 qui définissent l'atteignabilité faible. Les équations 11 à 13 sont des copies des équations 4 à 6 qui définissent les exigences sur la politique, où r_π est remplacé par wr_π. Dans l'équation 11, $df_\pi(o)$ est remplacé par la formulation équivalente $(\pi(o) \neq \perp)$. L'équation 10 est finalement simplifiée pour prendre en compte l'équation 11.

Alors que le CSP précédent P (voir la section 2.1) impliquait $|\mathbf{d}(O)| + |\mathbf{d}(S)| + |\mathbf{d}(S)|^2$ variables, ce CSP P' n'en n'implique plus que $|\mathbf{d}(O)| + |\mathbf{d}(S)|$.

Nous allons montrer que, comme P, P' modélise précisément le problème de synthèse de contrôleur, c'est-à-dire que l'approche consistant à résoudre P' pour résoudre le problème de synthèse de contrôleur est correcte, complète et possiblement optimale, dans un sens que nous allons définir.

2.4 Correction, complétude et optimalité

Correction La correction de l'approche résulte de la propriété 2. Supposons en effet que P' admette une solution, faite d'une politique π et d'une relation wr_π. D'après les équations 9 et 10, la relation wr_π est une relation d'atteignabilité faible. Soit r_π la relation d'atteignabilité associée à π. On a d'après la propriété 2 : $\forall s \in \mathbf{d}(S), r_\pi(s) \rightarrow wr_\pi(s)$. La relation r_π satisfait donc les exigences associées aux équations 11 à 13. En conséquence, π est une solution du problème de synthèse de contrôleur.

Complétude Quant à la complétude de l'approche, elle résulte de la propriété 3. Supposons en effet que le problème de synthèse de contrôleur admette une solution, sous la forme d'une politique π et d'une relation

d'atteignabilité associée r_π. D'après la propriété 3, r_π est une relation d'atteignabilité faible. π et r_π constituent donc une solution de P'. P' est donc cohérent et un solveur de contraintes complet produira une solution (celle-ci ou une autre).

Optimalité Il reste que, si P' admet une solution, la politique produite π peut être définie pour des observations non atteignables, qui ne sont associées à aucun état atteignable. Pratiquement, cela ne constitue pas un problème puisque ces observations ne seront jamais atteintes en suivant π. Cependant si, pour des raisons de lisibilité ou de compacité, on désire une politique π qui ne soit définie que pour les observations atteignables (associées à au moins un état atteignable), il suffit de transformer le problème de satisfaction de contraintes en un problème d'optimisation sous contraintes où le critère à minimiser est le nombre d'observations pour lesquelles la politique est définie ($\sum_{o\in\mathbf{d}(O)}(\pi(o)\neq\bot)$). Si l'on obtient une solution optimale avec preuve d'optimalité, on a d'après la propriété 4 la garantie que la relation wr_π produite est la relation d'atteignabilité r_π et que la politique π produite n'est définie que pour les observations atteignables. Sinon, on obtient une politique éventuellement définie pour des observations non atteignables, ce qui reste acceptable.

2.5 Modèle OPL

Nous avons utilisé le langage OPL [11] pour exprimer les modèles associés à divers problèmes de synthèse de contrôleur de systèmes dynamiques. N'importe quel autre langage de programmation par contraintes aurait pu être utilisé. Les modèles OPL générés sont optimisés de façon à limiter autant que possible le nombre de contraintes induites. Ils exploitent en particulier le fait que la relation $T(s,c,s')$ (respectivement $P(s,c,s')$) est habituellement une conjonction de relations $T_1(s)$ (respectivement $P_1(s)$), $T_2(s,c)$ (respectivement $P_2(s,c)$) et $T_3(s,c,s')$ (respectivement $P_3(s,c,s')$). La relation $T_1(s)$ (respectivement $P_1(s)$) exprime les états possibles (respectivement acceptables). La relation $T_2(s,c)$ (respectivement $P_2(s,c)$) exprime les commandes possibles (respectivement acceptables). Finalement, la relation $T_3(s,c,s')$ (respectivement $P_3(s,c,s')$) exprime les transitions possibles (respectivement acceptables). Pour le moment, ces modèles sont définis manuellement, ce qui est potentiellement une source d'erreurs. La construction automatique d'un modèle OPL à partir des ensembles de variables S, O et C et des ensembles de contraintes qui définissent les relations I, T et P est cependant envisageable.

Sur l'exemple jouet de la section 1.3, le modèle OPL génère 112 variables et 980 contraintes. Il est résolu par l'outil *CP Optimizer* associé à OPL en moins d'1/100 de seconde. La politique produite est la suivante :

$$w_N \wedge w_S \wedge \neg w_E \wedge \neg w_W : \ m = m_W$$
$$w_N \wedge \neg w_S \wedge \neg w_E \wedge w_W : \ m = m_S$$
$$\neg w_N \wedge w_S \wedge \neg w_E \wedge w_W : \ m = m_N$$

Chaque ligne est associée à une observation faiblement atteignable. Sur chaque ligne, l'observation apparaît avant les deux points et la commande associée après. Par exemple, la première ligne spécifie que, si des murs sont observés au nord et au sud, mais ni à l'est, ni à l'ouest, le robot doit se déplacer vers l'ouest. Cette politique consiste à se déplacer vers l'ouest, puis à alterner mouvements vers le nord et vers le sud. Dans ce cas, la politique produite sans optimisation explicite est par chance optimale : elle est définie uniquement pour les observations atteignables.

3 Un exemple réel

Pour valider l'approche proposée, nous l'avons mise en œuvre sur un exemple réel de contrôle d'un sous-système d'un satellite, précédemment introduit dans [16].

Le contexte est un satellite de surveillance de la Terre dont la mission est de détecter des points chauds à la surface de la Terre, dus à des feux de forêt ou à des éruptions volcaniques [6]. Il est équipé d'un instrument de détection à large fauchée et d'un instrument d'observation à fauchée réduite. En cas de détection de point chaud, une alarme doit être adressée au sol via un satellite géostationnaire relais et des observations de la zone doivent être réalisées. Les données d'observation sont ensuite télé-déchargées vers des stations sol lors de passages en visibilité. Dans ce contexte, nous nous intéressons à un équipement appelé DSP (*Digital Signal Processor*) en charge de l'analyse des images produites par l'instrument de détection et de la détection de points chauds dans ces images. Le DSP est composé de trois éléments :

— un analyseur, en charge de l'analyse des images ;
— un circuit, qui fournit le courant nécessaire à l'analyseur ;
— un interrupteur, qui permet d'activer ou désactiver la tension aux bornes du circuit.

Le module de contrôle du DSP reçoit des requêtes de mise ON ou OFF en provenance du module de contrôle de la fonction de détection. À chaque étape, il produit différentes sorties (voir la figure 4) :

— une commande de l'interrupteur du DSP ;

— des signaux vers le module de contrôle de la détection, indiquant si le DSP fonctionne correctement ou non ;
— un signal vers le module de contrôle de l'alarme, indiquant si, oui ou non, un point chaud a été détecté.

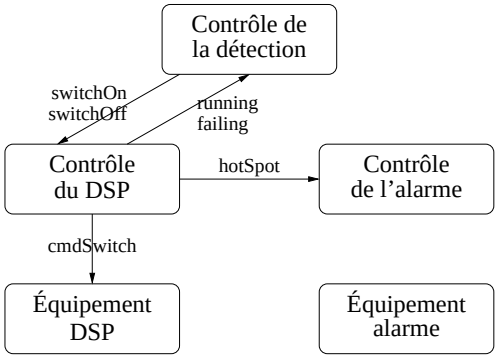


FIGURE 4 – Entrées et sorties du module en charge de contrôler le DSP

Chacun des trois éléments constituant le DSP est susceptible de tomber en panne. Si l'analyseur n'est pas en panne et reçoit du courant, la détection de points chauds est supposée fonctionner correctement. S'il est en panne ou ne reçoit pas de courant, la détection ne fonctionne pas. Si le circuit est en panne, l'analyseur ne reçoit pas de courant. Si l'interrupteur est en panne, la tension aux bornes du circuit n'est pas forcément conforme à la commande de l'interrupteur.

Informellement, les propriétés de sûreté de plus haut niveau qui doivent être satisfaites à chaque étape sont les suivantes : en absence de pannes, si le DSP est ON, la détection doit être correcte (détection de point chaud en cas de point chaud ; pas de détection, sinon) ; s'il est OFF, il ne doit rien détecter.

Variables d'état Pour modéliser ce problème, nous utilisons les variables d'état suivantes (ensemble S) :

— $switchOn \in \{0,1\}$: présence ou non d'une requête de mise ON : 1 pour présence ;
— $switchOff \in \{0,1\}$: présence ou non d'une requête de mise OFF : 1 pour présence ;
— $switched \in \{0,1\}$: dernière requête de mise ON ou OFF reçue par le contrôleur (mémoire du contrôleur) : 1 pour ON, 0 pour OFF ;
— $tension \in \{0,1\}$: présence ou non de tension aux bornes du circuit : 1 pour présence ;
— $current \in \{0,1\}$: présence ou non de courant dans le circuit : 1 pour présence ;
— $faultAnalyzer \in \{0,1\}$: panne ou non de l'analyseur : 1 pour panne ;
— $faultCircuit \in \{0,1\}$: panne ou non du circuit : 1 pour panne ;
— $faultSwitch \in \{0,1\}$: panne ou non de l'interrupteur : 1 pour panne ;
— $inputIm \in \{NOIM, NORM, HOT\}$: nature de l'image en entrée de l'analyseur : $NOIM$ en cas d'absence d'image, $NORM$ pour une image normale sans point chaud, HOT pour une image avec point chaud ; pour le contrôle du DSP, on ne s'intéresse pas au contenu des images, par exemple, à la position géographique des points chauds ; on s'intéresse uniquement à la présence ou non de points chauds ;
— $resultAnal \in \{NOIM, NORM, HOT\}$: résultat de l'analyse.

Parmi ces variables, seules les variables $switchOn$, $switchOff$, $switched$, $tension$, $current$ et $resultAnal$ sont observables par le contrôleur (ensemble O). Les pannes, ainsi que la nature réelle de l'image en entrée de l'analyseur, ne sont pas connues du contrôleur.

Variables de commande Nous utilisons par ailleurs les variables de commande suivantes (ensemble C) :

— $cmdSwitch \in \{0,1\}$: commande de l'interrupteur : 1 pour ON, 0 pour OFF ;
— $cmdMemory \in \{0,1\}$: mise à jour de la mémoire du contrôleur : 1 pour ON, 0 pour OFF ;
— $running \in \{0,1\}$: information de bon fonctionnement du DSP : 1 pour correct ;
— $failing \in \{0,1\}$: information de mauvais fonctionnement du DSP : 1 pour incorrect ;
— $hotSpot \in \{0,1\}$: information de détection de point chaud : 1 pour détection.

Graphe de dépendance Pour structurer l'écriture du modèle (relations I, T et P), il peut être utile de construire le graphe qui représente les dépendances entre variables d'état et de commande. La figure 5 montre ce graphe dans un cadre de représentation proche du langage graphique utilisé par l'outil SCADE [7]. Les boites marquées "pre" représentent l'accès à la valeur précédente d'une variable d'état. On y voit par exemple que la tension courante dépend de la tension précédente, de la commande de l'interrupteur et de son état (panne ou non).

États initiaux possibles Les états initiaux possibles (ensemble I) sont définies par les contraintes suivantes sur les variables d'état : $\neg switchOn$, $\neg switchOff$, $\neg switched$, $\neg current$, $\neg tension$, $(resultAnal = NOIM)$, $(inputIm = NOIM)$. Des pannes sont donc possibles dans l'état initial.

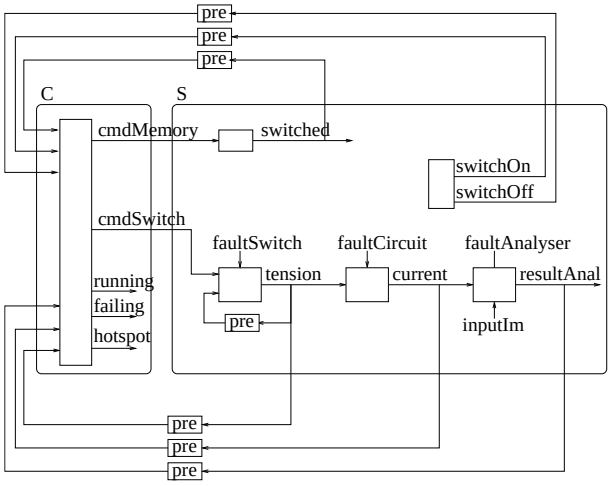


FIGURE 5 – Graphe de dépendances entre variables d'état et de commande

Transitions possibles Les transitions possibles (ensemble T) sont définies par les contraintes suivantes sur les variables d'état et de commande :

Contraintes sur les modules de plus haut niveau :
$\neg(switchOn \wedge switchOff)$

Contraintes sur la mémoire du contrôleur :
$switched' = cmdMemory$

Contraintes sur l'interrupteur :
$\neg tension \wedge \neg cmdSwitch \rightarrow \neg tension'$
$tension \wedge cmdSwitch \rightarrow tension'$
$\neg faultSwitch \rightarrow (tension' = cmdSwitch)$

Contraintes sur le circuit :
$current = (tension \wedge \neg faultCircuit)$

Contraintes sur l'analyseur :
$(\neg current \vee faultAnalyzer)$
$\quad \rightarrow (resultAnal = NOIM)$
$(current \wedge \neg faultAnalyzer)$
$\quad \rightarrow (resultAnal = inputIm)$

Rappelons que, pour chaque variable d'état x, x' représente sa valeur à l'étape suivante. Les contraintes sur les modules de plus haut niveau traduisent l'hypothèse qu'il ne peut pas y avoir simultanément une requête de mise ON et une autre de mise OFF. Les contraintes sur la mémoire du contrôleur expriment que la mémoire est toujours correctement mise à jour. Les contraintes sur l'interrupteur expriment qu'en l'absence de panne de l'interrupteur, la tension est conforme à la commande de l'interrupteur. Celles sur le circuit expriment qu'il y a du courant si et seulement si il y a de la tension et pas de panne du circuit. Enfin, celles sur l'analyseur expriment qu'en l'absence de panne de l'analyseur, le résultat de l'analyse est conforme à la nature de l'image en entrée.

Transitions acceptables Les transitions acceptables (ensemble P) sont définies par les contraintes suivantes sur les variables d'état et de commande :

Exigences sur la mise à jour de la mémoire du contrôleur :
$switchOn \rightarrow cmdMemory$
$switchOff \rightarrow \neg cmdMemory$
$(\neg switchOn \wedge \neg switchOff)$
$\quad \rightarrow (cmdMemory = switched)$

Exigences sur la détection de point chaud :
$hotSpot = (running \wedge (resultAnal = HOT))$

Exigences sur la détection de pannes :
$running = (switched \wedge (resultAnal \neq NOIM))$
$failing = (switched \wedge (resultAnal = NOIM)) \vee$
$\quad (tension \neq switched)) \vee (current \neq tension))$
$\neg(running \wedge failing)$
$switched \rightarrow (running \vee failing)$

Exigences de plus haut niveau :
$((inputIm = HOT) \wedge running) \rightarrow hotSpot$
$hotSpot \rightarrow (inputIm = HOT)$
$(\neg faultSwitch \wedge \neg faultCircuit' \wedge \neg faultAnalyser'$
$\quad \wedge switched') \rightarrow (resultAnal' = inputIm')$
$(\neg faultSwitch \wedge \neg switched')$
$\quad \rightarrow (resultAnal' = NOIM)$

Par exemple, les contraintes sur la détection de point chaud imposent qu'un signal de détection est émis si et seulement si le DSP fonctionne correctement et si le résultat de l'analyse indique la présence de point chaud. Les contraintes sur la détection de pannes imposent que le contrôleur du DSP n'indique pas simultanément qu'il est en bon et en mauvais fonctionnement

et que, si le DSP est ON, il indique l'un ou l'autre. Les exigences de plus haut niveau imposent que, si l'image en entrée contient un point chaud et si le DSP fonctionne correctement, alors un signal de détection est émis. Inversement, elles imposent que si un signal de détection est émis, l'image en entrée contient effectivement un point chaud. De plus, elles imposent qu'en l'absence de panne, si le DSP est ON, le résultat de l'analyse est conforme à la nature de l'image en entrée. Par contre, elles imposent qu'en l'absence de panne de l'interrupteur, si le DSP est OFF, il n'y a pas de résultat d'analyse.

Le modèle OPL associé génère 2356 variables et 512175 contraintes. Il est résolu par *CP Optimizer* en 3,39 secondes grâce à la propagation initiale qui affecte 2338 variables (résultats obtenus sur une station de travail SUN Ultra 45 fonctionnant sous Unix et utilisant un processeur de 1.6 GHz et 1 GB de RAM). Comme avec l'exemple jouet de la section 1.3, la politique produite sans optimisation explicite est par chance optimale : elle est définie uniquement pour les observations atteignables.

4 Forces et faiblesses de l'approche proposée

Dans ce papier, nous avons proposé une formulation du problème de synthèse de contrôleur pour un système dynamique comme un pur problème de satisfaction (optimisation) de contraintes. C'est à notre connaissance la première fois qu'une telle formulation est proposée. Au delà des applications classiques de la programmation par contraintes à la planification et à l'ordonnancement, ce travail ouvre la voie à l'application de la programmation par contraintes au vaste domaine du contrôle de systèmes dynamiques à évènements discrets. Il faut souligner que la formulation proposée peut être utilisée pour synthétiser ou pour vérifier des contrôleurs : la vérification est plus simple que la synthèse parce que la politique est connue en cas de vérification, alors qu'elle est inconnue en cas de synthèse.

L'approche proposée est à rapprocher de la formulation du problème de synthèse d'une politique optimale pour un MDP (*Markov Decision Process* [18]) comme un pur problème de programmation linéaire. En effet, dans cette formulation, une variable est associée à chaque état, représentant la valeur optimale qu'il est possible d'obtenir à partir de cet état. Les équations d'optimalité de Bellman [2] se traduisent par un ensemble de contraintes linéaires : une par paire état-commande. Le critère à minimiser est la somme des variables. L'atteignabilité n'est cependant pas pris en compte : tous les états sont supposés atteignables à toute étape. L'approche que nous avons proposée peut être vue comme l'équivalent "logique" de cette approche de résolution des MDP.

Le principal avantage de l'approche proposée est qu'elle permet, pour la résolution, de se reposer entièrement sur les outils génériques existants de programmation par contraintes. Quand les modèles de programmation par contraintes seront construits automatiquement à partir de la définition du problème (ensembles de variables S, O et C et ensembles de contraintes définissant les relations I, T et P), seule la définition du problème devra être modifiée pour passer d'un système dynamique à un autre.

Son défaut principal est le nombre de variables et de contraintes générées (au moins une variable par état et une contrainte par paire d'états). Comme le nombre d'états possibles est une fonction exponentielle du nombre de variables d'état (à ne pas confondre avec les variables CSP), l'approche ne pourra être mise en œuvre que sur les problèmes impliquant un nombre relativement faible de variables d'état (de l'ordre de la dizaine, comme dans l'exemple du DSP). On notera cependant que les algorithmes de synthèse de contrôleurs développés dans le domaine de l'automatique classique [9] produisent des résultats pour des systèmes dynamiques continus ayant un nombre de variables d'état du même ordre. D'autres approches à base de contraintes, plus économes en termes de variables et de contraintes, devraient cependant être explorées. L'approche précédemment proposée dans [13] est beaucoup plus économe en termes de variables et de contraintes. Malheureusement, elle est correcte, mais incomplète : le problème généré est sur-contraint par rapport au problème originel.

Finalement, dans ce papier, nous nous sommes focalisés sur la modélisation et nous ne nous sommes pas vraiment intéressés à l'efficacité de la résolution. Des comparaisons expérimentales en termes d'efficacité avec des algorithmes génériques dédiés à la synthèse de contrôleur devraient être menées.

Remerciements

Ce travail a été mené dans le cadre du projet français CNES-ONERA AGATA (Architecture Générique pour l'Autonomie : Tests et Applications ; voir `http://www.agata.fr`) dont l'objectif est de développer des techniques permettant d'accroître l'autonomie des systèmes spatiaux. Nous tenons à remercier Michel Lemaître pour ses précieuses contributions.

Références

[1] A. Arnold, A. Vincent, and I. Walukiewicz. Games for Synthesis of Controllers with Partial Information. *Theoretical Computer Science*, 303(1) :7–34, 2003.

[2] R. Bellman. *Dynamic Programming.* Princeton University Press, 1957.

[3] P. Bertoli, A. Cimatti, M. Roveri, and P. Traverso. Planning in Nondeterministic Domains under Partial Observability via Symbolic Model Checking. In *Proc. of the 17th International Joint Conference on Artificial Intelligence (IJCAI-01)*, pages 473–478, Seattle, WA, USA, 2001.

[4] B. Bonet, H. Palacios, and H. Geffner. Automatic Derivation of Memoryless Policies and Finite-State Controllers Using Classical Planners. In *Proc. of the 19th International Conference on Automated Planning and Scheduling (ICAPS-09)*, Thessaloniki, Greece, 2009.

[5] E. Clarke, O. Grumberg, and D. Peled. *Model Checking.* MIT Press, 1999.

[6] S. Damiani, G. Verfaillie, and M.-C. Charmeau. An Earth Watching Satellite Constellation : How to Manage a Team of Watching Agents with Limited Communications. In *Proc. of the 4th Conference on Autonomous Agents and Multi-Agent Systems (AAMAS-05)*, pages 455–462, Utrecht, The Netherlands, 2005.

[7] Esterel Technologies. SCADE. http ://www.esterel-technologies.com/.

[8] M. Ghallab, D. Nau, and P. Traverso. *Automated Planning : Theory and Practice.* Morgan Kaufmann, 2004.

[9] F. Golnaraghi and B. Kuo. *Automatic Control Systems.* John Wiley & Sons, 2009.

[10] N. Halbwachs. *Synchronous Programming of Reactive Systems.* Kluwer, 1993.

[11] IBM ILOG. Cplex optimization studio. http ://www-01.ibm.com/software/integration/optimization/cplex-optimization-studio/.

[12] B. Jobstmann, S. Galler, M. Weiglhofer, and R. Bloem. Anzu : A Tool for Property Synthesis. In *Proc. of the 19th International Conference on Computer Aided Verification (CAV-07)*, pages 258–262, Berlin, Germany, 2007.

[13] M. Lemaître, G. Verfaillie, C. Pralet, and G. Infantes. Synthèse de contrôleur simplement valide dans le cadre de la programmation par contraintes. In *Actes des Journées Françaises sur la Planification, la Décision et l'Apprentissage pour la Conduite de Systèmes (JFPDA-10)*, Besançon, France, 2010.

[14] MBP Team. Model Based Planner. http ://sra.itc.it/tools/mbp/.

[15] A. Pnueli and R. Rosner. On the Synthesis of a Reactive Module. In *Proc. of the ACM Symposium on Principles of Programming Languages (POPL-89)*, pages 179–190, 1989.

[16] C. Pralet, M. Lemaître, G. Verfaillie, and G. Infantes. Synthesizing Controllers for Autonomous Systems : A Constraint-based Approach. In *Proc. of the 10th International Symposium on Artificial Intelligence, Robotics, and Automation for Space (i-SAIRAS-10)*, Sapporo, Japan, 2010.

[17] C. Pralet, G. Verfaillie, M. Lemaître, and G. Infantes. Constraint-based Controller Synthesis in Non-deterministic and Partially Observable Domains. In *Proc. of the 19th European Conference on Artificial Intelligence (ECAI-10)*, pages 681–686, Lisbon, Portugal, 2010.

[18] M. Puterman. *Markov Decision Processes, Discrete Stochastic Dynamic Programming.* John Wiley & Sons, 1994.

[19] P. Ramadge and W. Wonham. The Control of Discrete Event Systems. *Proc. of the IEEE*, 77(1) :81–98, 1989.

[20] RATSY Team. RATSY : Requirements Analysis Tool with Synthesis. http ://rat.fbk.eu/ratsy/.

[21] F. Rossi, P. Van Beek, and T. Walsh, editors. *Handbook of Constraint Programming.* Elsevier, 2006.

Programmation par contraintes pour les problèmes de plus grand sous-graphe commun

Philippe Vismara
LIRMM, UMR5506 CNRS - Université Montpellier II, Montpellier, France
MISTEA, UMR729 Montpellier SupAgro - INRA, Montpellier, France
vismara@lirmm.fr

Résumé

Les problèmes de satisfaction de contraintes ont montré leur efficacité pour résoudre des questions de comparaison de graphes comme l'isomorphisme de graphe ou de sous-graphe. La recherche du plus grand sous-graphe commun est un problème plus difficile qui a été peu abordé par des CSP. Dans cet article nous identifions différentes déclinaisons de ce problème en termes de théorie des graphes (sous-graphe induit, partiel, connexe, ...). Puis nous étudions la possibilité de les modéliser dans un solver classique en abordant notamment les questions de connexité et de symétrie. Nous présentons enfin quelques résultats expérimentaux.

Abstract

Constraint Programming has proven its efficiency to solve graph matching problems such as graph or subgraph isomorphism. It is much harder to compute maximum common subgraphs and this problem has received little attention in the CSP litterature. In this paper we discuss different variants of the problem. We consider how to model them with a conventional CSP solver and we focus on the connexity and symmetry topics. Finally, we present some experimental results.

1 Introduction

Le recours de plus en plus fréquent aux modèles de graphes pour représenter des connaissances a fait grandir le besoin d'outils efficaces pour les manipuler. La plupart des problèmes de graphes étant NP complet, il n'est pas étonnant que de nombreux travaux en programmation par contraintes se soient récemment intéressés, avec succès, à des questions comme l'isomorphisme de graphe (non classé) ou de sous-graphe. La recherche du plus grand sous-graphe commun est un problème plus difficile qui a été peu abordé par des CSP alors qu'il se pose fréquemment dans des domaines comme la Chimie [24] ou la Reconnaissance de formes [4].

Dans cet article nous identifions les différentes déclinaisons de ce problème en termes de théorie des graphes (sous-graphe induit, partiel, connexe, ...) puis nous étudions leur modélisation dans le cadre de la programmation par contraintes.

Par soucis de clarté nous nous limiterons au cas des graphes non orientés bien que cette étude soit généralisable aux graphes orientés.

2 Les problèmes de plus grand sous-graphe commun

2.1 Notations

Étant donné un graphe simple non orienté $G = (N, E)$, $N = \{1, ...n\}$ désigne l'ensemble des n sommets de G et E l'ensemble des m arêtes.

Une arête entre x et y sera généralement notée xy (sachant que $xy = yx$).

Si le graphe est étiqueté, on notera $l_G : N \cup E \rightarrow \mathcal{L}$ la fonction qui associe à chaque sommet ou arête de G un label appartenant à l'ensemble $\mathcal{L}$ de tous les labels (de tous les graphes traités).

$\forall N' \subseteq N$, on note $E(N') = \{uv \in E \mid u, v \in N'\}$

Un *sous-graphe induit* de G est un graphe $G' = (N', E')$ tel que $N' \subseteq N$ et $E' = E(N')$.

Un *graphe partiel* de G est un graphe $G' = (N, E')$ tel que $E' \subseteq E$.

Un *sous-graphe partiel* de G est un graphe $G' = (N', E')$ tel que $N' \subseteq N$ et $E' \subseteq E(N')$.

Le *graphe des arêtes* (*line graph*) de G, est le graphe $L(G) = (E, \mathcal{E})$ tel que $\mathcal{E} = \{\alpha\beta \in E \times E \mid \alpha \cap \beta \neq \varnothing\}$.

2.2 Définitions

Dans la littérature, le terme de *plus grand sous-graphe commun* peut faire référence à des notions très différentes suivant qu'il s'agit par exemple de sous-graphes *induits* ou *partiels*. Cette confusion est encore plus importante dans les articles anglophones où "sub-graph" est souvent (mais pas toujours) synonyme de sous-graphe induit.

Étant donnés deux graphes $G_A = (N_A, E_A)$ et $G_B = (N_B, E_B)$, résoudre un problème de plus grand sous-graphe commun consiste à trouver une fonction partielle injective $\mu : N_A \longrightarrow N_B$ telle que :
MCIS (induit) : $dom(\mu)$ soit de taille maximum et $\forall x, y \in dom(\mu), xy \in E_A \Leftrightarrow \mu(x)\mu(y) \in E_B$
MCCIS (induit connexe) ≡ MCIS et le graphe $G' = (dom(\mu), E(dom(\mu)))$ est connexe.
MCS (partiel) : l'ensemble E' des arêtes préservées par μ soit de taille maximum avec $E' = \{xy \in dom(\mu) \times dom(\mu) \mid xy \in E_A \wedge \mu(x)\mu(y) \in E_B\}$.
MCCS (partiel connexe) ≡ MCS et le graphe $G(dom(\mu), E')$ est connexe.

Si les graphes sont étiquetés, μ doit conserver les labels : $\forall x \in dom(\mu), l_{G_A}(x) = l_{G_B}(\mu(x))$ et $\forall x, y \in dom(\mu), l_{G_A}(xy) = l_{G_B}(\mu(x)\mu(y))$

Ces définitions sont illustrées par la figure 1 qui donne un exemple de solution pour chacune d'elle.

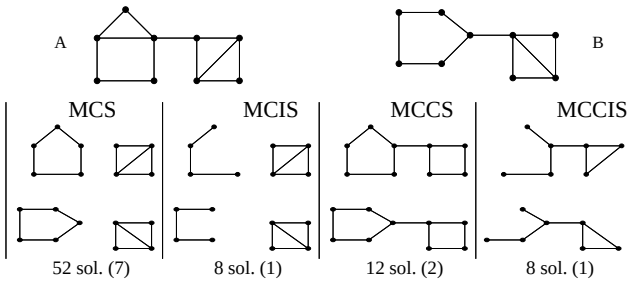


FIG. 1 – Différents plus grands sous-graphes communs

Le résultat obtenu est généralement très différent suivant la définition choisie, sans qu'il soit possible d'identifier une définition meilleure que les autres, y compris au sein d'un même champ d'application où ces notions permettent de modéliser des questions très diverses.

Par ailleurs, chaque problème admet souvent un grand nombre de solutions (52 MCS différents pour la figure 1). Or la recherche d'un plus grand sous graphe commun n'est souvent qu'une simplification d'un problème de comparaison de graphes beaucoup plus complexe. Si cette spécificité ne peut être formalisée, il devient nécessaire de confronter les résultats obtenus au regard d'un expert. Dans ce cas, il est indispensable d'éliminer les solutions symétriques. Par exemple, dans la figure 1, le nombre de solutions non symétriques est indiqué entre parenthèses.

D'un point de vue formel, nous dirons que deux solutions μ et μ' sont équivalentes à une symétrie de G_A (resp. G_B) près s'il existe un automorphisme σ_A (resp. σ_B) de G_A tel que $\mu' = \mu \circ \sigma_A$ (resp $\mu' = \sigma_B \circ \mu$).

2.3 Transformer un MCS en MCIS

La notion de *graphe des arêtes* fournit un moyen simple de transformer un problème de MCS en MCIS. La figure 2 donne un exemple d'équivalence entre les sous-graphes partiels de deux graphes et les sous-graphes induits correspondants dans leurs graphes des arêtes.

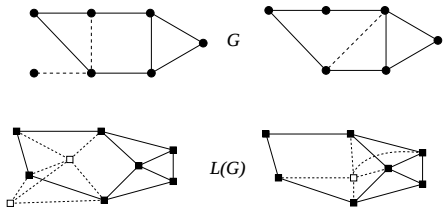


FIG. 2 – Transformer un MCS en MCIS du graphe des arêtes.

D'après le théorème de Withney (1935), deux graphes sont isomorphes si et seulement si leurs graphes des arêtes le sont, sauf si l'un des deux est K_3 et l'autre $K_{1,3}$ (interchange triangle / trinode). La figure 3 donne un exemple d'interchange trinode/triangle, si on considère uniquement les arêtes a, b et c. Par contre, les graphes A et B en entier sont bien isomorphes.

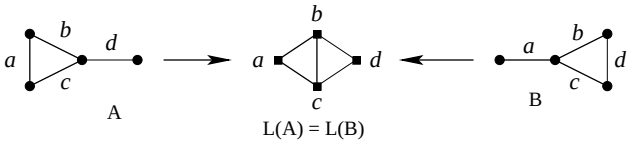


FIG. 3 – Interchange triangle / trinode (en gras)

En s'appuyant sur la preuve de ce théorème dans [11], on peut montrer que si μ est solution du MCIS entre G_A et G_B, on peut déduire de μ une unique solution du MCS entre $L(G_A)$ et $L(G_B)$ et réciproquement, pour peu que $|dom(\mu)| \geq 5$. Dans la figure 3, le morphisme $\mu(a) = a$, ..., $\mu(d) = d$ est une solution du MCIS($L(A)$, $L(B)$) mais pas une solution de MCS(A,B). Par contre, il suffit d'ajouter un sommet aux graphes A et B pour qu'un tel interchange soit impossible.

Du point de vue de la complexité, cette transformation MCS → MCIS ne simplifie pas le problème puisqu'un graphe des arêtes $L(G)$ contiendra $|E|$ sommets.

Par ailleurs il faut veiller à interdire les interchanges triangles / trinodes, surtout pour la version du problème non nécessairement connexe.

Dans le cas de graphes étiquetés, l'équivalence sera préservée si les sommets de $L(G)$ sont étiquetés en combinant le label de l'arête de G correspondante avec ceux de ces extrémités dans G ; une arête de $L(G)$ sera quant à elle étiquetée par le label du sommet correspondant dans G.

La *subdivision* est une autre façon de transformer un MCS en MCIS. Le graphe subdivisé $S(G) = (N \cup E, \mathcal{H})$ est obtenu en insérant un sommet entre les extrémités de chaque arête (cf. fig. 4). Dans ce cas, tout sous-graphe partiel de G devient un sous-graphe induit de $S(G)$ dans lequel chaque noeud issu de E possède 0 ou 2 voisins. Le graphe $S(G)$ contient donc $|E| + |N|$ sommets ce qui le rend généralement moins intéressant que le graphe des arêtes.

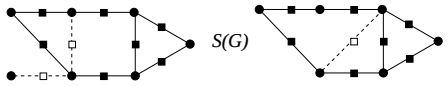


FIG. 4 – Graphe subdivisé

2.4 État de l'art

Comparé à d'autres problèmes de graphes, la recherche d'un plus grand sous-graphe a été très peu traitée dans la littérature. Dans certains domaines comme la Chimie, différentes heuristiques ont parfois été proposées sans qu'il soit possible d'en mesurer l'efficacité ou la correction. C'est pourquoi nous nous limiterons ici à présenter les deux principaux algorithmes utilisés.

La première méthode, probablement la plus répandue, consiste à transformer un MCIS en la recherche de cliques dans un graphe de compatibilité $G_{\mathcal{C}} = (N_{\mathcal{C}}, E_{\mathcal{C}})$. Ce graphe est défini sur l'ensemble des paires de sommets compatibles des deux graphes : $N_{\mathcal{C}} = \{(a, b) \in N_A \times N_B \mid l_{G_A}(a) = l_{G_B}(b)\}$. Deux sommets sont reliés dans $G_{\mathcal{C}}$ si les couples qu'ils représentent respectent la connexité des graphes G_A et G_B : $E_{\mathcal{C}} = \{(a_1, b_1)(a_2, b_2) \in N_{\mathcal{C}} \times N_{\mathcal{C}} \mid a_1a_2 \in E_A \Leftrightarrow b_1b_2 \in E_B \wedge l_{G_A}(a_1a_2) = l_{G_B}(b_1b_2)\}$.

Toute clique de $G_{\mathcal{C}}$ correspond naturellement à un morphisme qui respecte la définition du MCIS. Le problème du MCIS se ramène alors à rechercher des cliques de taille maximum dans $G_{\mathcal{C}}$ [18]. Cette approche est par exemple utilisée dans [10]. Il existe de nombreux travaux sur la détermination des cliques maximum [1] y compris dans le cadre de la programmation par contraintes [26]. Pour résoudre le problème du MCCIS, Koch [14] propose une variante de l'algorithme de Bron et kerbosch [2] afin d'obtenir des cliques de $G_{\mathcal{C}}$ qui correspondent à un sous-graphe connexe dans G_A ou G_B.

Le second type de méthodes regroupe toutes les approches qui ne construisent pas de graphe de compatibilité mais explorent directement l'espace de recherche avec un mécanisme de "backtrack". L'algorithme de Mc Gregor [19], initialement conçu pour résoudre un MCS, est un des plus cité pour résoudre un MCIS. Cet algorithme utilise une démarche analogue à la programmation par contraintes qui s'apparente au *forward checking*. Plus récemment, un autre algorithme de backtrack a été proposé par [15] avec là aussi une approche très similaire au *forward checking* et une optimisation de l'exploration qui revient à choisir, à chaque étape, la variable dont le domaine est de taille minimum.

Les deux types d'approches que nous venons d'évoquer ont été comparées dans [5] sur une base de 81400 paires de graphes aléatoires ayant différentes caractéristiques (densité, régularités, ...). Leur conclusion est qu'aucun des algorithmes étudiés ne se distingue clairement, le classement s'inversant suivant le type de graphe. On peut également remarquer que bon nombre d'algorithmes d'énumération de cliques ont une approche de type "backtrack". Tant que le graphe de compatibilité n'est pas trop grand, les deux types d'algorithmes peuvent donc être amenés à explorer des espaces de recherche très semblables.

Par ailleurs, nous avons vu que plusieurs algorithmes dédiés au MCIS emploient des techniques très analogues à celles utilisées en programmation par contraintes. Ce paradigme n'est pourtant jamais évoqué dans la littérature sur les problèmes de plus grand sous-graphe commun. La prochaine section s'intéresse donc à la résolution de ces problèmes par un CSP.

3 Un CSP pour déterminer le plus grand sous-graphe commun

Ces dernières années, plusieurs problèmes d'appariement de graphes ont fait l'objet d'une modélisation par un réseau de contraintes. Qu'il s'agisse d'isomorphisme de sous-graphe [25, 28, 16, 37, 29] ou d'isomorphisme de graphe [30], ces travaux reposent sur un modèle dans lequel les variables correspondent à des sommets des graphes traités. Une approche plus originale a été proposée dans [9] pour modéliser des problèmes de sous-graphe dans CP(Graph) [8] puis d'appariement de graphes dans CP(Map)[35]. Dans ce modèle, les variables sont des graphes ou des fonctions d'appariement entre ces graphes. Cette approche a montré son efficacité pour résoudre des problèmes comme l'isomorphisme de sous-graphe [7] mais n'a pas été généralisée aux problèmes de sous-graphe commun.

Plus récemment, un formalisme général a été proposé dans [17] pour modéliser les principaux problèmes d'appariement de graphes (graph matching) incluant les problèmes cités précédemment ainsi que le MCS et le MCIS. Cet élégant modèle considère un *matching* comme un sous-ensemble de $N_A \times N_B$ sur lequel il est possible de définir une série de contraintes de base. Les problèmes d'appariement peuvent alors être construits en combinant ces différentes contraintes. Ce modèle a été mis en oeuvre dans Comet [31], un langage de programmation qui combine contraintes et recherche locale. Les résultats ont montré l'efficacité de cette approche pour résoudre l'isomorphisme de sous-graphe. Quant au problème du MCS, il a été implémenté par un algorithme tabou permettant d'obtenir une solution approchée en temps raisonnable (quelques dizaines de secondes) pour des graphes de taille moyenne (50 sommets) extraits de la base de tests de [5].

La mise au point d'approches efficaces pour déterminer des plus grands sous-graphes communs semble donc toujours d'actualité. En attendant l'émergence d'outils dédiés à ces problèmes, il est intéressant de voir dans quelle mesure il est déjà possible d'en résoudre certains, dans le cadre classique de la programmation par contraintes. La suite de cette section a pour but d'identifier les principales questions à résoudre et de proposer une modélisation de ces problèmes. L'objectif n'est pas de trouver la meilleure modélisation mais d'en proposer une suffisamment simple pour être rapidement mise en oeuvre.

3.1 Sous-graphes induits (MCIS)

Pour résoudre un MCIS, entre $G_A = (N_A, E_A)$ et $G_B = (N_B, E_B)$, on peut construire un CSP dont les variables correspondent aux sommets de G_A ou de G_B ou même des deux. Nous allons utiliser cette seconde solution qui facilite la modélisation du problème, notamment pour la recherche de solutions connexes. Par ailleurs elle évite d'introduire une distinction arbitraire entre les deux graphes.

L'ensemble des variables du CSP se décompose donc en deux sous-ensembles : $X_A = \{a_i\}_{i \in N_A}$ pour N_A et $X_B = \{b_j\}_{j \in N_B}$ pour N_B.

On notera $n_A = |N_A|$ (resp $n_b = |N_B|$) et on supposera que $N_A = 1..n_a$.

Pour obtenir une fonction partielle et injective, on associe une valeur spéciale, propre à chaque variable, afin d'identifier les sommets perdus :

$$\forall i \in 1..n_A,\ D(a_i) = N_B \cup \{n_B + i\}$$

Si les graphes G_A et G_B sont étiquetés, on ne garde que les valeurs compatibles :

$$D(a_i) = \{k \in N_B \mid l_{G_B}(k) = l_{G_A}(i)\} \cup \{n_B + i\}$$

Pour résoudre le MCIS, on pose les contraintes suivantes :

channeling on définit une contrainte de channeling partiel entre les variables de X_A et X_B, $\forall a_i \in X_A, b_j \in X_B, (a_i \leq n \wedge a_i = j) \Leftrightarrow (b_j \leq n \wedge b_j = i)$

différence puisque chaque variable de X_A ou de X_B possède sa propre valeur spéciale, on obtiendra une solution injective en définissant une contrainte de différence $allDifferent(a_1, ..., a_{n_A})$ et $allDifferent(b_1, ..., b_{n_B})$

voisinage l'isomorphisme des sous-graphes induits est garanti par $n_A + n_B$ contraintes binaires :
- $\forall ik \in E_A, (a_i \leq n_B \wedge a_k \leq n_B) \Rightarrow a_i a_k \in E_B$
- $\forall jz \in E_B, (b_j \leq n_A \wedge b_z \leq n_A) \Rightarrow b_j b_z \in E_A$

Si les graphes sont étiquetés, ces contraintes deviennent :
- $\forall ik \in E_A, (a_i \leq n \wedge a_k \leq n) \Rightarrow (a_i a_k \in E_B \wedge l_{G_A}(ik) = l_{G_B}(a_i a_k))$
- $\forall jz \in E_B, (b_j \leq n \wedge b_z \leq n) \Rightarrow (b_j b_z \in E_A \wedge l_{G_B}(jz) = l_{G_A}(b_j b_z))$

Pour obtenir une solution de taille maximale, on définit une variable *taille* qui compte le nombre de sommets conservés dans G_A et G_B (donc $D(taille) = 1..(n_A + n_B)$). Une contrainte *among* permet de garantir que *taille* est le nombre de variables dont la valeur est inférieure à n_A ou n_B.

3.2 Sous-graphes induits connexes (MCCIS)

On trouve dans la littérature peu de méthodes pour définir une contrainte de connexité.

Dans CP(Graph)[8] deux solutions ont été étudiées pour la recherche de sous-graphes d'un graphe donné (sans morphisme). La première repose sur la fermeture transitive de la relation de connexité. Elle suppose de définir $O(n^3)$ contraintes disjonctives N-aires et $O(n^4)$ variables booléennes, ce qui rend cette solution inadaptée aux grands graphes. La seconde est une contrainte globale qui réalise le filtrage par un simple parcours en profondeur du graphe ($O(m + n)$ en temps par filtrage). Dès qu'au moins un sommet est conservé dans le sous-graphe, on peut éliminer tous les sommets qui ne sont pas dans la même composante connexe. On peut aussi conserver tous les points d'articulation[1] (et les isthmes[2]) situés entre deux sommets conservés dans la solution courante.

L'algorithme utilisé pour ce filtrage n'est pas détaillé mais il reste facile à concevoir à partir d'un parcours en profondeur (Tarjan), avec calcul d'un ordre de *pre* et *post* d'exploration des sommets. En partant d'un sommet d tel que $a_d \leq n_B$, un point d'articulation sera

[1] point d'articulation = sommet dont la suppression déconnecte le graphe

[2] isthme = arête dont la suppression déconnecte le graphe

conservé si le descendant qui a permis de le détecter faisait partie d'une composante 2-connexe contenant elle aussi un sommet s tel que $a_s \leq n_B$.

Une contrainte globale similaire est proposée dans [20] pour le calcul d'un graphe partiel (tous les sommets sont conservés). Un parcours en profondeur est utilisé pour déterminer un arbre recouvrant du graphe. Cet arbre est stocké dans une structure de données réversible avec une série de compteurs indiquant le nombre de cycles fondamentaux[3] passant par chaque arête. Cette structure (en $O(n^2)$) permet de détecter la déconnexion du graphe partiel sans lancer un parcours à chaque filtrage (sauf en cas de suppression d'une arête de l'arbre).

Une autre façon de garantir la connexité de la solution [32], mais sans filtrage supplémentaire, est d'utiliser un ordre d'instanciation des variables qui choisisse la nouvelle variable à instancier parmi les voisines des variables déjà instanciées (avec une valeur $a_s \leq n_B$). Une contrainte globale permet de maintenir à jour la composante connexe en cours de construction ainsi que l'ensemble des variables voisines non instanciées (nécessite des structures de données réversibles de taille $O(n)$). Lorsque ce voisinage devient vide, il suffit de vérifier que toutes les variables instanciées font partie de la composante connexe.

3.3 Éliminer les symétries

Dès qu'il s'agit d'énumérer toutes les solutions, il est indispensable d'éliminer celles qui sont équivalentes à un automorphisme de G_A ou de G_B près. Cette élimination peut être réalisée a posteriori mais avec un coût important. Par ailleurs, éliminer ces symétries au cours de la résolution peut fortement limiter l'espace de recherche dans la mesure où les symétries du CSP sont généralement équivalentes à celles des graphes traitées. Il est cependant possible de supprimer les solutions symétriques d'un CSP sans nécessairement éliminer toutes les branches symétriques de l'arbre de recherche.

Une des démarches les plus courantes pour éliminer les symétries de variables consiste à poser des contraintes lexicographiques [6]. Étant donné un ordre $x_{i_1}, \dots x_{i_n}$ sur les variables, on pose, pour chaque symétrie de variable σ, une contrainte de la forme $x_{i_1}, \dots x_{i_n} \leq_{lex} x_{\sigma(i_1)}, \dots x_{\sigma(i_n)}$. Malheureusement, il peut exister un nombre exponentiel de symétries.

Cette même méthode peut être utilisée pour éliminer toute symétrie de valeur θ en posant une contrainte de la forme $x_{i_1}, \dots x_{i_n} \leq_{lex} \theta(x_{i_1}), \dots \theta(x_{i_n})$ [33]. Mais maintenir la GAC est dans ce cas NP-Complet [34]. Quant aux combinaisons de symétries entre symétrie de variables σ et symétrie de valeur θ, on peut les éliminer en posant des contraintes de la forme $x_{i_1}, \dots x_{i_n} \leq_{lex} \theta(x_{\sigma(i_1)}), \dots \theta(x_{\sigma(i_n)})$ [23].

Pour la recherche d'un isomorphisme de sous-graphe de G_p dans G_t, les auteurs de [36] ont montré qu'il est possible d'éliminer les symétries du graphe G_p, sur lequel sont définies les variables x_i, en ajoutant un nombre linéaire de contraintes. Cette méthode utilise les résultats de [22] sur l'élimination des symétries de variables dans les problèmes injectifs. Elle nécessite le calcul du groupe $\mathcal{A}_{G_p}$ des automorphismes de G_p (par exemple avec le logiciel NAUTY) et l'algorithme de Schreier-Sims pour construire une chaîne de stabiliseurs. Pour chaque sommet i de G_P tel que l'ensemble $\mathcal{S}_i = \{j \neq i \mid \exists \sigma \in \mathcal{A}_{G_p} \text{ tq } i = \sigma(j) \wedge \forall k < j, \sigma(k) = k\}$ n'est pas vide, on définit $r(i) = \max(j \in \mathcal{S}_i)$. On peut noter que $r(i) < i$. Pour éliminer toutes les symétries de $\mathcal{A}_{G_p}$, il suffit de poser une contrainte de la forme $x_{r(i)} < x_i$ pour chaque $r(i)$ ainsi défini.

Cette méthode peut être utilisée pour éliminer les symétries du CSP permettant de résoudre un MCIS. Si on considère, l'ensemble X_A des variables liées à G_A et les contraintes binaires définies pour chaque arête de E_A, on obtient un sous-problème de contraintes dont les symétries sont équivalentes à celles de G_A. On peut alors éliminer les symétries de ce CSP par des contraintes de la forme $a_{r_A(i)} < a_i$, avec $r_A(i) < i$ obtenu comme précédemment à partir du groupe d'automorphismes $\mathcal{A}_{G_A}$. Ces contraintes sont compatibles avec les valeurs spéciales de la forme $i + n_B$ ajoutées aux domaines $D(a_i)$.

Le même raisonnement peut être employé pour éliminer les symétries de G_B en considérant le sous-problème correspondant aux variables de X_B. Les contraintes seront de la forme $b_{r_B(j)} < b_j$.

Malheureusement, il n'est pas possible d'utiliser cette méthode en même temps sur G_A et sur G_B. Par exemple, dans la figure 5, la méthode proposée par Puget [22] permet d'éliminer les symétries de G_A en posant l'unique contrainte $a_1 < a_3$. De même, les symétries de G_B sont éliminées par la contrainte $b_1 < b_2$. Or le CSP n'admet que deux solutions qui chacune respecte ou viole une des deux contraintes :

- $a_0 = 0,\ a_1 = 3,\ a_2 = 2,\ a_3 = 4,\ a_4 = 1$
  $b_0 = 0,\ b_1 = 4,\ b_2 = 2,\ b_3 = 1,\ b_4 = 3$
- $a_0 = 0,\ a_1 = 4,\ a_2 = 1,\ a_3 = 3,\ a_4 = 2$
  $b_0 = 0,\ b_1 = 2,\ b_2 = 4,\ b_3 = 3,\ b_4 = 1$

On peut remarquer que l'élimination des symétries de variables du CSP réduit à X_B est équivalente à l'élimination ses symétries de valeurs du CSP réduit à X_A. Nous sommes donc confrontés au problème de l'élimination conjointe des symétries de variables et de

[3] étant donné un arbre recouvrant $T = (N, E')$ de $G = (N, E)$ toute arête de $E \setminus E'$ ajoutée à T forme un seul cycle dit *fondamental*

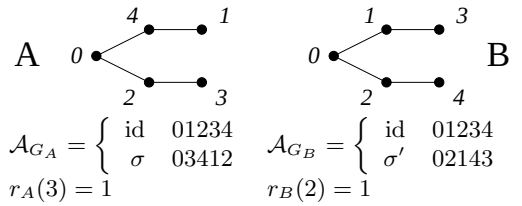


FIG. 5 – Un contre-exemple

valeurs d'un CSP.

Nous avons vu que la méthode proposée par Puget dans [22] permet d'éliminer les symétries de variables liées par un AllDiff, en posant un nombre linéaire de contraintes. L'auteur a montré que cette approche est compatible avec la méthode GE-Tree décrite dans [27] pour éliminer les symétries de valeurs. Cette méthode nécessite le calcul, à chaque noeud de l'arbre de recherche, des symétries de valeurs invariantes ($\theta(v) = v$) pour les valeurs déjà affectées. Ces symétries induisent des classes d'équivalence sur les valeurs restant dans le domaine des variables non instanciées. L'instanciation de la prochaine variable est alors limitée à un seul représentant de chacune de ces classes, éliminant ainsi les branches symétriques.

Dans [21], Puget propose d'éliminer les symétries de valeurs dans les problèmes surjectifs. Il définit un ensemble $\{z_j\}$ de variables supplémentaires, en plus des variables $\{x_i\}$ du CSP, et une contrainte de channeling qui s'écrit : $(x_i = j) \leftrightarrow (z_j = i)$ dans le cas d'un AllDiff sur les $\{x_i\}$. Les symétries de valeurs du CSP étant équivalentes aux symétries des variables z_j, on peut les supprimer en posant un nombre linéaire de contraintes comme dans [22].

L'analogie avec les variables a_i et b_i du CSP MCIS décrit au paragraphe 3.1 est évidente et le contre-exemple de la figure 5 s'applique aussi. Il a d'ailleurs été démontré dans [13], que l'utilisation conjointe de contraintes lexicographiques pour éliminer les symétries de variables avec la méthode de Puget [21] pour supprimer les symétries de valeurs peut faire perdre des solutions non symétriques. Cette incompatibilité interdit d'éliminer les symétries de variables et de valeurs du MCIS en utilisant, dans les deux cas, la méthode de Puget pour ne poser qu'un nombre linéaire de contraintes.

Au final, il n'est donc pas toujours possible d'éliminer toutes les symétries de façon efficace. Le choix de la méthode employée dépendra fortement de la taille des groupes de symétrie de chaque graphe.

3.4 Sous-graphes partiels (MCS)

Une première méthode pour résoudre un MCS consiste à le considérer comme un MCIS dans lequel les contraintes binaires d'adjacence peuvent être violées [17].

Si on reste dans le cadre d'un CSP plus classique, on peut résoudre le MCS en se ramenant à un MCIS sur les graphes des arêtes, grâce à la transformation présentée au paragraphe 2.3.

Pour trouver un MCS connexe (MCCS) entre G_A et G_B il suffit de chercher un MCCIS entre $L(G_A)$ et $L(G_B)$. En dehors du cas trivial de l'interchange de la figure 3, toute solution contenant au moins 5 sommets n'admet pas d'interchange.

Dans le cas d'un MCS quelconque (non-connexe), le nombre d'interchanges possibles est beaucoup plus important et une vérification a posteriori ne garantit pas une solution maximale. On doit donc ajouter des contraintes pour interdire les interchanges. Sans le channeling, il faudrait définir une contrainte pour chaque triplet d'arêtes afin de garantir qu'un trinode ou triangle sera préservé par la fonction d'appariement. Grâce au channeling, il suffit de créer une contrainte ternaire par triangle de chacun des deux graphes G_A et G_B, l'énumération initiale des cycles de taille 3 étant réalisée en temps linéaire [3].

4 Résultats expérimentaux et discussion

Le modèle de MCIS présenté dans cet article a été implémenté en Java en utilisant le solveur CHOCO [12] qui fournit la plupart des contraintes nécessaires. Seule la contrainte de *partial channeling* a du être adaptée à partir de la contrainte *Channeling* présente dans CHOCO.

La contrainte de connexité à été implémentée par une contrainte globale réalisant le parcours en profondeur évoqué au paragraphe 3.2.

L'élimination des symétries a été réalisée en combinant la librairie NAUTY et une implementation non optimisée de l'algorithme de Schreier-Sims.

Quant au MCS, il a été programmé en appliquant le MCIS sur le graphe des arêtes et en y ajoutant les contraintes ternaires interdisant un interchange.

Cette étape d'implémentation a montré qu'il est tout a fait possible de programmer un problème de plus grand sous-graphe commun avec un solveur comme CHOCO. La partie la plus complexe n'est d'ailleurs pas liée aux CSP mais à la mise au point des algorithmes de détection des symétries.

Pour évaluer l'efficacité de ces modèles, nous avons utilisé la base de graphes aléatoires construite par [5] pour le problème du MCIS. Cette base contient des séries homogènes de 100 couples de graphes entièrement étiquetés (une étiquette différente par sommets), de 10 à 100 sommets. Sur ces graphes, tous les problèmes modélisés ont été résolus en moins d'une secondes avec

un processeur à 3 Ghz. Pour réduire le nombre d'étiquettes (par exemple à la moitie du nombre de sommets), les auteurs de la base de données proposent de remplacer chaque étiquette (entre 0 et 2^{16}) par sa valeur modulo le nombre de valeurs souhaitées. Mais cette réduction ne garantit pas d'obtenir le nombre de labels voulus, notamment au niveau des arêtes et il est difficile d'exploiter les résultats. Nous avons donc choisi de comparer les graphes en supprimant toutes les étiquettes.

D'une manière globale, il existe une très grande disparité de temps de calcul, même au sein d'une même classe de graphes. Les résultats préliminaires qui suivent sont donc à interpréter avec prudence.

Les tableaux ci-dessous concernent la classe "mcs90_r005" de graphes peu denses de 20, 25 et 30 sommets et dont les couples ont 90% de parties communes. Étant peu denses, ces graphes ont un nombre d'arêtes proches du nombre de sommets ce qui permet d'avoir des instances de tailles semblables pour les versions "induit" et "partiel" des problèmes.

	MCCIS		
n	*GAC*	*BC*	*Clique*
20	$0,9$	$0,5$	$4,3$
25	$9,6$	$4,5$	$26,1$
30	$134,5$	$61,8$	> 1000

Pour la contrainte AllDiff, nous avons testé la différence entre les versions GAC et BC, cette dernière semblant plus rapide.

Dans le cas du MCCIS, la méthode reposant sur la recherche de cliques dans le graphe de compatibilité a été implémentée, en utilisant la variante de l'algorithme de Bron et Kerbosh proposée par [14]. Les résultats sont moins bons que ceux obtenus par les CSP. Il serait interessant de vérifier si des algorithmes récents de recherche de cliques ne permettraient pas de réduire cet écart.

Le tableau suivant montre que les temps de calcul obtenus pour le MCCIS ou le MCCS sont meilleurs que ceux des versions non connexes, ce qui est cohérent avec la combinatoire inhérente à ces problèmes. On peut également en conclure que la méthode de filtrage utilisée pour la contrainte de connexité, bien que perfectible, est déjà utilisable en l'état.

n	m	*MCIS*		*MCCS*		*MCS*	
		GAC	*BC*	*GAC*	*BC*	*GAC*	*BC*
20	21	$6,7$	$4,3$	$2,69$	$1,7$	$12,9$	$8,7$
25	29	$67,1$	$45,7$	$120,0$	$104,9$	375	348

L'élimination des symétries a été testée en appliquant la méthode de [22] sur un seul des graphes comparés (celui ayant le plus grand nombre d'automorphismes). Malheureusement, les graphes testés possèdent pas ou peu de symétries et si les temps de calculs sont meilleurs, ils ne sont pas significatifs.

Au final, on peut s'interroger sur la pertinence de la base de données de tests, notamment du fait de l'impossibilité de tenir compte des étiquettes qui sont souvent présentes dans des problèmes du monde réel. La grande variabilité des résultats obtenus sur des graphes théoriquement semblables pose également problème.

5 Conclusion

Dans cet article nous avons proposé une modélisation du problème de plus grand sous graphe commun dans le cadre classique de la programmation par contraintes. Les premières expérimentations sur des graphes aléatoires ont montré l'efficacité de cette démarche sur des instances assez simples mais mériteraient d'être poursuivies pour en mesurer les limites.

Cela étant, il serait probablement plus intéressant d'expérimenter ce modèle sur des problèmes réels pour lesquels des contraintes complémentaires pourraient être identifiées et formalisées. Il serait alors possible de disposer d'instances de problèmes réels pour mieux évaluer les différentes approches actuellement développées par la communauté de la programmation par contraintes pour résoudre les MCIS et autres MCS.

Références

[1] I. M. Bomze, M. Budinich, P. M. Pardalos, and M. Pelillo. The maximum clique problem. In D.-Z. Du and P. M. Pardalos, editors, *Handbook of Combinatorial Optimization (Supplement Volume A)*, pages 1–74. Kluwer Academic, 1999.

[2] C. Bron and J. Kerbosch. Finding all cliques of an undirected graph. *Communication of the ACM*, 16(9) :575–579, 1973.

[3] N. Chiba, T. Nishizeki, S. Abe, and T. Ozawa. A linear algorithm for embedding planar graphs using PQ-trees. *Journal of computer and system sciences*, 30 :54–76, 1985.

[4] D. Conte, P. Foggia, C. Sansone, and M. Vento. Thirty years of graph matching in pattern recognition. *IJPRAI*, 18(3) :265–298, 2004.

[5] D. Conte, P. Foggia, and M. Vento. Challenging complexity of maximum common subgraph detection algorithms : A performance analysis of three algorithms on a wide database of graphs. *Journal of Graph Algorithms and Applications*, 11(1) :99–143, 2007.

[6] J. Crawford, M. Ginsberg, E. Luks, and A. Roy. Symmetry-breaking predicates for search problems. In *KR*, pages 148–159. Morgan Kaufmann, 1996.

[7] Y. Deville, G. Dooms, and S. Zampelli. Combining two structured domains for modeling various

graph matching problems. In *Proceedings CSCLP 2007*, volume 5129 of *Lecture Notes in Computer Science*, pages 76–90, 2008.

[8] G. Dooms. *The CP(Graph) Computation Domain in Constraint Programming*. PhD thesis, Université catholique de Louvain, 2006.

[9] G. Dooms, Y. Deville, and P. Dupont. Cp(graph) : Introducing a graph computation domain in constraint programming. In *Proc CP 2005*. Springer Verlag, 2005.

[10] Paul J. Durand, Rohit Pasari, Johnnie W. Baker, and Chun che Tsai. An efficient algorithm for similarity analysis of molecules. *Internet Journal of Chemistry*, 2(17), June 1999.

[11] F. Harary. *Graph Theory*. Addison-Wesley, 1969.

[12] Narendra Jussien, Guillaume Rochart, and Xavier Lorca. Choco : an Open Source Java Constraint Programming Library. In *CPAIOR'08 Workshop on Open-Source Software for Integer and Contraint Programming (OSSICP'08) CPAIOR'08 Workshop on Open-Source Software for Integer and Contraint Programming (OSSICP'08)*, pages 1–10, Paris, France France, 2008.

[13] G. Katsirelos, N. Narodytska, and T. Walsh. On the complexity and completeness of static constraints for breaking row and column symmetry. In *Proceedings of the 16th international conference on Principles and practice of constraint programming*, CP'10, pages 305–320. Springer-Verlag, 2010.

[14] I. Koch. Enumerating all connected maximal common subgraphs in two graphs. *Theoretical Computer Science*, 250 :1–30, 2001.

[15] Evgeny B. Krissinel and Kim Henrick. Common subgraph isomorphism detection by backtracking search. *Software : Practice and Experience*, 34(6) :591–607, 2004.

[16] J. Larossa and G. Valiente. Constraint satisfaction algorithms for graph pattern matching. *Math. Struct. Comput. Sci.*, 12(4) :403–422, 2002.

[17] V. le Clément, Y. Deville, and C. Solnon. Constraint-based graph matching. In *Proc. Principles and Practice of Constraint Programming - CP 2009*, volume 5732 of *Lecture Notes in Computer Science*, pages 274–288, 2009.

[18] G. Levi. A note on the derivation of maximal common subgraphs of two directed or undirected graphs. *Calcolo*, 9(4) :341–352, 1972.

[19] James J. McGregor. Backtrack search algorithm and the maximal common subgraph problem. *Software Practice and Experience*, 12 :23—34, 1981.

[20] P. Prosser and C. Unsworth. A connectivity constraint using bridges. *Frontiers in Artificial Intelligence and Applications*, 141 :707, 2006.

[21] J.-F. Puget. Breaking all value symmetries in surjection problems. In *Proceedings of the 11th international conference on Principles and practice of constraint programming*, CP'05, pages 490–504, 2005.

[22] J.-F. Puget. Breaking symmetries in all different problems. In *IJCAI*, pages 272–277, 2005.

[23] J.-F. Puget. An efficient way of breaking value symmetries. In *AAAI*, 2006.

[24] J. W. Raymond and P. Willett. Maximum common subgraph isomorphism algorithms for the matching of chemical structures. *Journal of Computer-Aided Molecular Design*, 16(7) :521–533, 2002.

[25] J.-C. Régin. *Développement d'outils algorithmiques pour l'Intelligence Artificielle et application Ã la chimie organique*. Thèse de doctorat, Université Montpellier II, 1995.

[26] J-C. Régin. Using constraint programming to solve the maximum clique problem. In *Principles and Practice of Constraint Programming - CP 2003, LNCS 2833*, pages 634–648. Springer, 2003.

[27] C. M. Roney-Dougal, I. P. Gent, T. Kelsey, and S. Linton. Tractable symmetry breaking using restricted search trees. In R. López de Mántaras and L. Saitta, editors, *ECAI*, pages 211–215. IOS Press, 2004.

[28] M. Rudolf. Utilizing constraint satisfaction techniques for efficient graph pattern matching. In *6th International Workshop on Theory and Application of Graph Transformations*, volume 1764 of *Lecture Notes in Computer Science*, pages 381–394. Springer-Verlag, 1998.

[29] C. Solnon. Alldifferent-based filtering for subgraph isomorphism. *Artificial Intelligence*, 174(12-13) :850 – 864, 2010.

[30] Sébastien Sorlin and Christine Solnon. A parametric filtering algorithm for the graph isomorphism problem. *Constraints*, 13 :518–537, 2008.

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

[32] Philippe Vismara and Benoît Valery. Finding maximum common connected subgraphs using clique detection or constraint satisfaction algorithms. In *Modelling, Computation and Optimization in Information Systems and Management Sciences, MCO 2008 Proceedings*, volume 14 of

Communications in Computer and Information Science, pages 358–368. Springer, 2008.

[33] Toby Walsh. General symmetry breaking constraints. In *CP*, pages 650–664, 2006.

[34] Toby Walsh. Breaking value symmetry. In *Proceedings of the 13th international conference on Principles and practice of constraint programming*, CP'07, pages 880–887. Springer-Verlag, 2007.

[35] S. Zampelli. *A Constraint Programming Approach to Subgraph Isomorphism.* PhD thesis, Université catholique de Louvain, juin 2008.

[36] S. Zampelli, Y. Deville, M. R. Saïdi, and B. Benhamou. Symmetry breaking in subgraph isomorphism. In *SymCon'07, the Seventh International Workshop on Symmetry and Constraint Satisfaction Problem. A Satellite Workshop of CP 2007.*, 2007.

[37] S. Zampelli, Y. Deville, and C. Solnon. Solving subgraph isomorphism problems with constraint programming. *Journal of Constraints*, 2010.

Avec le soutien de