Dixièmes Journées Francophones de Programmation par Contraintes

JFPC 2014

Angers, 11 – 13 Juin 2014

JFPC 2014

Actes des Dixièmes Journées Francophones de Programmation par Contraintes

à l'initiative de l'Association Française pour la Programmation par Contraintes (AFPC)

Organisation

LERIA, Université d'Angers

Président des Journées

Frédéric Lardeux, LERIA, Université d'Angers

Président du comité de programme

Thierry Petit, TASC, INRIA, CNRS-LINA, Mines de Nantes

Comité d'organisation

Frédéric Lardeux, LERIA, Université d'Angers (président)
Matthieu Basseur, LERIA, Université d'Angers
Caner Candan, LERIA, Université d'Angers
Fabien Chhel, LERIA, Université d'Angers
Adrien Goëffon, LERIA, Université d'Angers
David Lesaint, LERIA, Université d'Angers
Eric Monfroy, LINA, Université de Nantes
Frédéric Saubion, LERIA, Université d'Angers

Comité de programme

Thierry Petit, TASC, INRIA / CNRS-LINA / Mines de Nantes, France (président)
Gilles Audemard, CRIL, Université d'Artois, France
Marc Christie, Université de Rennes 1/INRIA, France
Rémi Coletta, LIRMM, Université de Montpellier, France
Emmanuel Coquery, Université Claude Bernard Lyon 1, France
Thi-Bich-Hanh Dao, LIFO, Université d'Orléans, France
Redouane Ezzahir, Université Ibnzohr, Maroc
Jean-Guillaume Fages, TASC, INRIA / CNRS-LINA / Mines Nantes, France
Djamal Habet, LSIS, Aix-Marseille Université, France
Marie-José Huguet, Université de Toulouse, INSA, LAAS-CNRS, France
George Katsirelos, INRA, Toulouse, France
Mathias Kleiner, Arts & Métiers, CNRS-LSIS, France
Arnaud Lallouet, GREYC, Université de Caen - Basse-Normandie, France
Frédéric Lardeux, LERIA, Université d'Angers, France
Nadjib Lazaar, LIRMM, Université de Montpellier, France
David Lesaint, LERIA, Université d'Angers, France
Chu-Min Li, MIS, Université de Picardie Jules Verne, France
Jean-Baptiste Mairy, Université catholique de Louvain, Belgique
Arnaud Malapert, I3S, Université de Nice Sophia Antipolis, France
Thierry Martinez, Inria Paris–Rocquencourt, France
Jean-Philippe Métivier, GREYC - Université de Caen Basse-Normandie, France
Bertrand Neveu, LIGM, Ecole des Ponts Paristech, France
Marie Pelleau, Université de Montréal, Canada
Laurent Péridy, IMA, Angers, France
Charles Prud'Homme, EMN Nantes, TASC, INRIA / CNRS-LINA / Mines Nantes, France
Philippe Refalo, IBM, France
Florian Richoux, LINA, TASC, INRIA / CNRS-LINA / Université de Nantes, France
Pierre Roy, Sony CSL Paris, France
Michel Rueher, Université de Nice Sophia Antipolis / I3S - CNRS, France
Sébastien Tabary, CRIL, Université d'Artois, France
Gilles Trombettoni, LIRMM, Université de Montpellier, France
Charlotte Truchet, LINA, Université de Nantes, France
Michel Vasquez, LGI2P, École des mines d'Alès, France
Nadarajen Veerapen, University of Stirling, Écosse, Royaume-Uni
Gérard Verfaillie, ONERA, Toulouse, France
Julien Vion, UVHC/LAMIH CNRS, France
Mohamed Wahbi, INSIGHT, University College Cork, Irlande
Stéphane Zampelli, PhD, Ecole Polytechnique de Louvain, Belgique
Matthias Zytnicki, INRA MIAT, France

Relecteurs additionnels

Patrice Boizumault, Adel Ferdjoukh, Benoît Hoessen, Claude Michel, Sylvain Piechowiak.

Préface

Les Journées Francophones de Programmation par Contraintes (JFPC) constituent le principal congrès de la communauté francophone travaillant sur la programmation par contraintes (PPC), les problèmes de satisfaction de contraintes (CSP), de satisfiabilité (SAT) et de programmation logique avec contraintes (CLP). Les JFPC regroupent aussi des thématiques liées en recherche opérationnelle, sur les méta-heuristiques, l'analyse par intervalles, etc. Pour la quatrième année consécutive, les JFPC sont organisées conjointement avec les Journées d'Intelligence Artificielle Fondamentale, offrant ainsi aux participants un spectre scientifique encore plus vaste.

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. Dixième édition de la série, les JFPC 2014 à Angers succèdent à celles d'Aix-en-Provance (2013), Toulouse (2012), Lyon (2011), Caen (2010), Orléans (2009), Nantes (2008), Rocquencourt (2007), Nîmes (2006) et Lens (2005).

L'objectif majeur des JFPC est de permettre aux scientifiques, industriels et étudiants francophones de se rencontrer, d'échanger et de présenter leurs travaux dans un cadre convivial. Cette année, les JFPC ont eu le plaisir de recevoir 46 soumissions provenant de Belgique, Canada, France, Irlande, Maroc, Tunisie, etc. Chaque article soumis a fait l'objet de relectures soignées par trois membres du comité de programme.

À l'issue des discussions menées sur la base des rapports, 38 articles ont été acceptés pour être présentés et publiés dans les actes, ce qui correspond à un taux d'acceptation de 82 %. Sachant qu'une part conséquente des soumissions est constituée de versions en français d'articles de conférences et de journaux internationaux parmi les plus sélectifs du domaine, ce taux est révélateur de la qualité scientifique des JFPC.

Le comité de programme a réuni 39 scientifiques académiques et industriels, issus de 6 pays différents, représentant plus de 30 organismes. Je remercie très sincèrement les membres du comité et les relecteurs additionnels pour la pertinence et la qualité de leur travail, ainsi que les auteurs qui nous ont permis de proposer un programme scientifique très attractif. Je remercie l'Association Française pour la Programmation par Contraintes (AFPC) pour son investissement régulier en termes d'animation et de promotion de la communauté. Enfin, je remercie vivement l'équipe d'organisation, les participants et les sponsors sans qui cet événement n'aurait pu avoir lieu.

Thierry Petit,
Président du comité de programme des JFPC 2014

Exposés invités

Gilles Chabert[1] et **Gilles Trombettoni**[2]

Programmation par contraintes sur intervalles pour l'optimisation globale et l'estimation de paramètres robuste

[1] LINA, Ecole des Mines de Nantes, France
[2] LIRMM, Université Montpellier 2, CNRS, France

Hadrien Cambazard

Contraintes NP-Difficiles avec des coûts : exemples d'applications et de filtrage

G-SCOP, Université de Grenoble / Grenoble INP / UJF-Grenoble / CNRS

Programmation par contraintes sur intervalles pour l'optimisation globale et l'estimation de paramètres robuste

Gilles Chabert[1] **Gilles Trombettoni**[2]

[1] LINA, Ecole des Mines de Nantes, France
[2] LIRMM, Université Montpellier 2, CNRS, France
Gilles.Chabert@mines-nantes.fr Gilles.Trombettoni@lirmm.fr

Résumé

Ou comment des algorithmes issus de l'informatique (programmation par contraintes, intelligence artificielle) offrent une alternative prometteuse aux mathématiques appliquées pour des problèmes numériques tels que l'optimisation globale continue et l'estimation de paramètres non-linéaire robuste.

Les intervalles donnent une représentation simple et élégante d'ensembles continus de points d'un espace de recherche. A l'origine utilisés dans le calcul scientifique, ils ont permis, depuis la fin des années 1980, le développement de méthodes de programmation par contraintes sur les réels. Ces méthodes se déclinent concrètement suivant deux types d'algorithmes : les *contracteurs* et les *extracteurs* de région intérieure.

Les contracteurs sont des algorithmes de filtrage (réduisant un domaine à un sous-domaine) vus comme des opérateurs mathématiques purs, c'est-à-dire, indépendants de la notion de contrainte et sans effet de bord. Cette vision abstraite permet de constituer une algèbre de contracteurs où la construction de stratégies complexes s'obtient par simple composition ; on parle alors de *programmation par contracteurs* [6].

La sémantique d'un contracteur se définit en fonction de la partie filtrée ; typiquement, celle-ci ne contient aucune solution du CSP. L'extraction de région intérieure est une opération duale à la contraction dans le sens où la sémantique est cette fois liée au sous-domaine calculé. Les algorithmes existants construisent notamment des boîtes ou des polytopes dits *intérieurs*, c'est-à-dire, contenant seulement des points solutions [3, 8].

Nous présentons dans une première partie la programmation par contracteurs et les principes de plusieurs contracteurs ; certains maintenant classiques comme l'algorithme de propagation de contraintes numériques Hc4 [4], d'autres plus récents comme Mohc [1, 7] qui exploitent plus finement des propriétés de monotonie. Nous présenterons également des contracteurs issus d'autres disciplines, telles que le traitement d'image (contracteur "pixel appartenant à une zone d'intérêt") ou les mathématiques discrètes (opérateur de q-intersection, liés au problème de la recherche de q-cliques dans un graphe).

Nous présentons ensuite deux méthodes d'extraction de régions intérieures originales [2] : une méthode duale à Hc4 pour l'extraction de boîtes intérieures et une méthode utilisant une formule de Taylor sur intervalles convexe pour l'extraction de polytopes intérieurs.

Dans la suite de l'exposé, nous montrerons comment ces opérateurs peuvent être combinés pour former des stratégies de résolution efficaces pour des problèmes fondamentaux tels que :

— L'optimisation globale sous contraintes, où notre stratégie IbexOpt [10] s'avère particulièrement efficace pour les problèmes non convexes. L'extraction de régions intérieures en optimisation globale sous contraintes offre en particulier un outil complémentaire à l'optimisation locale permettant d'obtenir de bons points réalisables. Les succès croissants de cette stratégie suggèrent que la PPC sur intervalles devient un paradigme important à l'interface de l'informatique et des mathématiques appliqués.

— L'estimation de paramètres robuste, où le contracteur de q-intersection joue un rôle clé pour la prise en compte des mesures aberrantes [5]. La stratégie d'estimation [9] basée sur la q-intersection offre une alternative aux méthodes statistiques classiques comme les moindres carrés ou la méthode du khi 2, ou encore à la méthode statistique RANSAC très utilisée en vision.

Tous les opérateurs et les stratégies décrits au cours de cet exposé sont implantés et disponibles dans la bibliothèque libre en `C++` IBEX (`www.ibex-lib.org`).

Références

[1] I. Araya, G. Trombettoni, and B. Neveu. Exploiting Monotonicity in Interval Constraint Propagation. In *Proc. AAAI*, pages 9–14, 2010.

[2] I. Araya, G. Trombettoni, B. Neveu, and G. Chabert. Upper Bounding in Inner Regions for Global Optimization under Inequality Constraints. *J. Global Optimization (JOGO)*, 10 :22 pages, 2014. to appear, online in 2014.

[3] F. Benhamou and F. Goualard. Universally Quantified Interval Constraints. In *Proc. CP, Constraint Programming, LNCS 1894*, pages 67–82, 2004.

[4] F. Benhamou, F. Goualard, L. Granvilliers, and J.-F. Puget. Revising Hull and Box Consistency. In *Proc. ICLP*, volume 5649 of *LNCS*, pages 230–244. Springer, 1999.

[5] C. Carbonnel, G. Trombettoni, P. Vismara, and G. Chabert. Q-intersection Algorithms for Constraint-Based Robust Parameter Estimation. In *AAAI*, page to appear. AAAI Press, 2014.

[6] G. Chabert and L. Jaulin. Contractor Programming. *Artificial Intelligence*, 173 :1079–1100, 2009.

[7] G. Chabert and L. Jaulin. Hull Consistency Under Monotonicity. In *Proc. CP, LNCS 5732*, pages 188–195, 2009.

[8] H. Collavizza, F. Delobel, and M. Rueher. Extending Consistent Domains of Numeric CSP. In *Proc. IJCAI*, pages 406–413, 1999.

[9] L. Jaulin, E. Walter, and O. Didrit. Guaranteed Robust Nonlinear Parameter Bounding. In *CESA'96 IMACS Multiconference (Symposium on Modelling, Analysis and Simulation)*, pages 1156–1161, 1996.

[10] G. Trombettoni, I. Araya, B. Neveu, and G. Chabert. Inner Regions and Interval Linearizations for Global Optimization. In *Proc. AAAI*, pages 99–104, 2011.

Contraintes NP-Difficiles avec des coûts: exemples d'applications et de filtrage

Hadrien Cambazard

G-SCOP, Université de Grenoble / Grenoble INP / UJF-Grenoble / CNRS
{hadrien.cambazard}@grenoble-inp.fr

Nous nous intéressons à la prise en compte pratique, en Programmation par Contraintes, de sous-problèmes NP-Difficiles avec des coûts. En particulier, nous discuterons de l'utilisation de la Relaxation Lagrangienne et la Programmation Dynamique pour mettre au point des mécanismes de filtrage.

De plus en plus de contraintes globales NP-Difficiles sont mises au point (BIN-PACKING, NVALUE, etc.) et intègrent des modèles de coûts. Ces contraintes s'appuient souvent sur la résolution de relaxations [5, 8] et la Relaxation Lagrangienne apparaît comme une technique de choix qui a été beaucoup utilisée en Recherche Opérationnelle [6, 9]. Elle peut permettre d'obtenir ou d'approximer la borne de la relaxation linéaire efficacement sans passer par un solveur linéaire [10] mais pose des difficultés pour une implémentation générique.

Le problème de l'*acheteur voyageur* qui constitue une généralisation du *voyageur de commerce* servira de fil conducteur. Nous essayerons néanmoins de donner un aperçu d'autres contextes applicatifs [2, 10, 3] ainsi que des contraintes pour lesquelles ces techniques ont été employées : MULTI-COST REGULAR [7], WEIGTED CIRCUIT [1], PMEDIAN [4].

Références

[1] P. Benchimol, W. van Hoeve, J.C. Régin, LM. Rousseau, and M. Rueher. Improved filtering for weighted circuit constraints. *Constraints*, 17(3) :205–233, 2012.

[2] H. Cambazard, D. Mehta, B. O'Sullivan, and H. Simonis. Bin packing with linear usage costs - an application to energy management in data centres. In Christian Schulte, editor, *CP*, volume 8124 of *LNCS*, p. 47–62. Springer, 2013.

[3] H. Cambazard, E. O'Mahony, and B. O'Sullivan. A shortest path-based approach to the multileaf collimator sequencing problem. *Discrete Applied Mathematics*, 160(1-2) :81–99, 2012.

[4] H. Cambazard and B. Penz. A constraint programming approach for the traveling purchaser problem. In Michela Milano, editor, *CP*, volume 7514 of *LNCS*, p. 735–749. Springer, 2012.

[5] F. Focacci, A. Lodi, and M. Milano. Embedding relaxations in global constraints for solving tsp and tsptw. *Ann. Math. Artif. Intell.*, 34(4) :291–311, 2002.

[6] M. L. Fisher. The lagrangian relaxation method for solving integer programming problems. *Management Science*, 50(12-Supp) :1861–1871, 2004.

[7] J. Menana and S. Demassey. Sequencing and counting with the multicost-regular constraint. In Willem Jan van Hoeve and John N. Hooker, editors, *CPAIOR*, volume 5547 of *LNCS*, p. 178–192. Springer, 2009.

[8] P. Refalo. Linear formulation of constraint programming models and hybrid solvers. In Rina Dechter, editor, *CP*, volume 1894 of *LNCS*, p. 369–383. Springer, 2000.

[9] M. Sellmann. Theoretical foundations of cp-based lagrangian relaxation. In Wallace [11], p. 634–647.

[10] M. Slusky and W. van Hoeve. A lagrangian relaxation for golomb rulers. In Carla P. Gomes and Meinolf Sellmann, editors, *CPAIOR*, volume 7874 of *LNCS*, p. 251–267. Springer, 2013.

[11] Mark Wallace, editor. *Principles and Practice of Constraint Programming - CP 2004, 10th International Conference, CP 2004, Toronto, Canada, September 27 - October 1, 2004, Proceedings*, volume 3258 of *LNCS*. Springer, 2004.

Articles acceptés

Combiner les Redémarrages, Nogoods et Décompositions pour la Résolution de CSP *

Philippe Jégou **Cyril Terrioux**

Aix-Marseille Université, LSIS UMR 7296
Avenue Escadrille Normandie-Niemen
13397 Marseille Cedex 20 (France)
{philippe.jegou, cyril.terrioux}@lsis.org

Résumé

Du point de vue théorique, les méthodes exploitant des décompositions (arborescentes) constituent une approche pertinente pour la résolution de CSP quand la largeur (arborescente) des réseaux de contraintes est petite. Dans ce cas, elles ont souvent montré leur intérêt pratique. Cependant, parfois, un mauvais choix pour le cluster racine (une décomposition arborescente est un arbre de clusters) peut dégrader significativement les performances de résolution.

Dans cet article, nous mettons en avant une explication de cette dégradation et nous proposons une solution reposant sur les techniques de redémarrage. Ensuite, nous présentons une nouvelle version de l'algorithme BTD (pour Backtracking with Tree-Decomposition [8]) intégrant des techniques de redémarrage. D'un point de vue théorique, nous prouvons que des nld-nogoods réduits peuvent être mémorisés et exploités durant la recherche et que leur taille est plus restreinte que celle des nld-nogoods enregistrés par MAC+RST+NG [9]. Nous décrivons également comment les (no)goods structurels peuvent être exploités quand la recherche redémarre à partir d'une nouvelle racine. Enfin, nous montrons empiriquement l'intérêt de l'approche proposée.

1 Introduction

Un problème de satisfaction de contraintes (CSP, voir [14] pour un état de l'art) est un triplet (X, D, C), où $X = \{x_1, \ldots, x_n\}$ est un ensemble de n variables, $D = (d_{x_1}, \ldots, d_{x_n})$ est une liste de domaines finis, à raison d'un domaine par variable, et $C = \{C_1, \ldots, C_e\}$ est un ensemble fini de e contraintes. Chaque contrainte C_i est un couple $(S(C_i), R(C_i))$, où $S(C_i) = \{x_{i_1}, \ldots, x_{i_k}\} \subseteq X$ est la *portée* de C_i, et $R(C_i) \subseteq d_{x_{i_1}} \times \cdots \times d_{x_{i_k}}$ est sa *relation de compatibilité*. L'*arité* de C_i est $|S(C_i)|$. Un CSP est dit *binaire* si toutes ses contraintes sont d'arité 2. La structure d'un réseau de contraintes est représentée par un hypergraphe (un graphe dans le cas binaire), appelé *hypergraphe de contraintes*, dont les sommets correspondent aux variables et les arêtes aux portées des contraintes. Dans cet article, pour simplifier le propos, nous ne traitons que du cas binaire. Cependant ce travail peut facilement s'étendre au cas non-binaire en exploitant la 2-section [1] (aussi appelée graphe primal) de l'hypergraphe de contraintes, ce que nous ferons d'ailleurs pour nos expérimentations lors desquelles nous considérerons des CSP binaires et non-binaires. De plus, sans perte de généralité, nous supposons que le graphe de contraintes est connexe. Pour simplifier les notations, dans la suite, nous noterons le graphe $(X, \{S(C_1), \ldots S(C_e)\})$ par (X, C). Une affectation sur un sous-ensemble de X est dite *cohérente* si elle ne viole aucune contrainte. Déterminer si un CSP possède une *solution* (i.e. une affectation cohérente de toutes les variables) est connu pour être NP-complet. Aussi, de nombreux travaux ont été réalisés pour rendre la résolution plus efficace en pratique en utilisant des algorithmes énumératifs optimisés qui peuvent exploiter des heuristiques, de l'apprentissage de contraintes, du retour en arrière non-chronologique, des techniques de filtrage basées sur la propagation de contraintes, etc. La complexité en temps de ces approches est naturellement exponentielle, au moins en $O(n.d^n)$ avec n le nombre de variables et d la taille du plus grand domaine.

Généralement, afin de garantir une résolution effi-

*Ce travail est soutenu par l'Agence Nationale de la Recherche dans le cadre du projet TUPLES (ANR-2010-BLAN-0210).

cace, la plupart des solveurs exploitent simultanément plusieurs techniques parmi celles précédemment citées (comme les heuristiques, l'apprentissage de contraintes ou le filtrage). De plus, souvent, ils tirent également profit de l'utilisation de techniques de redémarrage. En particulier, les techniques de redémarrage permettent habituellement de réduire l'impact de mauvais choix effectués par des heuristiques (comme l'heuristique d'ordonnancement des variables) ou de l'apparition du phénomène de *queue lourde*. Elles ont été récemment introduites dans le cadre des CSP (e.g. dans [9]). Pour des raisons liées à l'efficacité, elles sont généralement exploitées avec des techniques d'apprentissage (comme la mémorisation de nld-nogoods [9]).

Dans cet article, nous introduisons pour la première fois, les techniques de redémarrage dans des méthodes de résolution de CSP par décomposition. Les méthodes par décomposition (e.g. [4, 8]) résolvent les CSP en prenant en compte certaines propriétés du réseau de contraintes. Souvent, elles reposent sur la notion de décomposition arborescente de graphes [12]. Dans ce cas, leur intérêt est relatif à leur complexité théorique en temps, i.e. $d^{w^+ + 1}$ avec w^+ la largeur de la décomposition arborescente considérée. Comme le calcul d'une décomposition arborescente optimale est NP-Difficile, les décompositions employées sont généralement produites à l'aide de méthodes heuristiques et donc fournissent une approximation d'une décomposition optimale. Quand le graphe possède de bonnes propriétés topologiques, et donc quand w^+ est petit, ces méthodes permettent de résoudre de grandes instances, e.g. les instances d'allocation de fréquence [3]. D'un point de vue pratique, elles obtiennent des résultats prometteurs sur de telles instances. Cependant, leur efficacité peut être significativement dégradée par de mauvais choix effectués par des heuristiques. Afin de présenter ce problème, nous considérerons ici la méthode BTD [8] qui constitue une référence dans l'état de l'art pour ce type d'approche [11].

Au niveau de BTD, la décomposition arborescente considérée et le choix d'un cluster racine (i.e. du premier cluster étudié) induisent un ordre particulier sur les variables. Aussi, sachant l'impact significatif que l'ordre sur les variables peut avoir sur l'efficacité d'un solveur, ce choix de cluster racine est crucial. Dans [7], l'approche proposée consiste à choisir l'ordre sur les variables avec plus de liberté, mais l'efficacité dépend toujours du choix du cluster racine. Dans la prochaine section, nous expliquons pourquoi il est difficile de proposer un choix de cluster racine convenable. En conséquence, afin de réduire l'impact du choix de racine sur l'efficacité pratique, nous proposons une alternative basée sur les techniques de redémarrage. Nous présentons alors une nouvelle version de BTD utilisant ces techniques. D'un point de vue théorique, nous prouvons ensuite que des nld-nogoods réduits peuvent être enregistrés durant la recherche et que leur taille est restreinte par rapport à celle des nld-nogoods mémorisés par MAC+RST+NG [9]. Nous décrivons aussi comment les (no)goods structurels peuvent être exploités quand la recherche redémarre à partir d'un nouveau cluster racine. Enfin, nous montrons expérimentalement l'intérêt de l'approche proposée.

La section 2 rappelle le cadre de BTD et décrit l'algorithme BTD-MAC [1]. Ensuite, la section 3 présente l'algorithme BTD-MAC+RST. Dans la section 4, nous évaluons l'intérêt des redémarrages pour la résolution de CSP à l'aide de méthodes par décomposition, avant de conclure dans la section 5.

2 La méthode BTD

La méthode BTD [8] constitue une référence dans l'état de l'art lorsqu'il s'agit de résoudre des CSP en exploitant la structure de leur graphe de contraintes [11]. Elle repose sur la notion de décomposition arborescente de graphes [12].

Définition 1 *Une* décomposition arborescente *d'un graphe $G = (X, C)$ est un couple (E, T) avec un arbre $T = (I, F)$ et une famille $E = \{E_i : i \in I\}$ de sous-ensembles de X, tel que chaque sous-ensemble (appelé cluster) E_i est un nœud de T et vérifie :*

(i) $\cup_{i \in I} E_i = X$,

(ii) pour chaque arête $\{x, y\} \in C$, il existe $i \in I$ avec $\{x, y\} \subseteq E_i$, et

(iii) pour tout $i, j, k \in I$, si k est sur un chemin de i à j dans T, alors $E_i \cap E_j \subseteq E_k$.

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

Etant donnés une décomposition arborescente (E, T) et un cluster racine E_r, nous notons $Desc(E_j)$ l'ensemble des sommets (variables) appartenant à l'union des descendants E_k de E_j dans le sous-arbre enraciné en E_j, E_j inclus. La figure 1(b) présente un arbre dont les nœuds correspondent aux cliques maximales du graphe de la figure 1(a). Il s'agit d'une décomposition arborescente possible pour ce graphe. Aussi, nous avons $E_1 = \{x_1, x_2, x_3\}$, $E_2 = \{x_2, x_3, x_4, x_5\}$, $E_3 = \{x_4, x_5, x_6\}$, $E_4 = \{x_3, x_7, x_8\}$, $Desc(E_1) = X$ et $Desc(E_2) = \{x_2, x_3, x_4, x_5, x_6\}$.

1. Cet algorithme n'a jamais été décrit précisément dans la littérature. L'algorithme MAC-BTD évoqué dans [8] correspond en fait à BTD-RFL (i.e. BTD basé sur Real-Full Lookahead).

Comme la taille maximale des clusters est 4, la largeur arborescente de ce graphe est 3.

Etant donné un ordre compatible $<$ sur les cluster (i.e. un ordre qui peut être produit par un parcours en profondeur d'abord de T à partir du cluster racine E_r), BTD effectue une recherche énumérative en utilisant un ordre $\preceq$ sur les variables (dit *compatible*) tel que $\forall x \in E_i$, $\forall y \in E_j$, avec $E_i < E_j$, $x \preceq y$. Autrement dit, l'ordre sur les clusters induit un ordre partiel sur les variables puisque les variables de E_i sont affectées avant celles de E_j si $E_i < E_j$. Pour l'exemple de la figure 1, $E_1 < E_2 < E_3 < E_4$ (resp. $x_1 \preceq x_2 \preceq x_3 \preceq \ldots \preceq x_8$) est un ordre compatible possible sur E (resp. X). En pratique, l'algorithme BTD débute sa recherche en affectant de façon cohérente les variables du cluster racine E_r avant d'explorer un cluster fils. Quand il explore un nouveau cluster E_i, il n'affecte que les variables de $E_i \backslash (E_i \cap E_{p(i)})$ [2].

Afin de résoudre chaque cluster, BTD peut exploiter n'importe quel algorithme qui ne modifie pas la structure du graphe de contraintes. Par exemple, BTD peut reposer sur l'algorithme MAC (pour Maintaining Arc-Consistency [15]). Durant la résolution, MAC peut prendre deux types de décisions : des *décisions positives* $x_i = v_i$ qui affectent la valeur v_i à la variable x_i (nous notons $Pos(\Sigma)$ l'ensemble des décisions positives de la suite de décisions Σ) et des *décisions négatives* $x_i \neq v_i$ qui assurent que x_i ne pourra pas être affectée à la valeur v_i. Soit $\Sigma = \langle \delta_1, \ldots, \delta_i \rangle$ (où chaque δ_j peut être une décision positive ou négative) la suite de décisions courante. Une nouvelle décision positive $x_{i+1} = v_{i+1}$ est choisie et un filtrage par cohérence d'arc (AC) est accompli. Si aucun domaine ne devient vide, la recherche continue avec la prise d'une nouvelle décision positive. Sinon, la valeur v_{i+1} est supprimée du domaine $d_{x_{i+1}}$, et un filtrage par AC est réalisé. Si un échec se produit à nouveau, MAC revient en arrière et change la dernière décision positive $x_\ell = v_\ell$ en $x_\ell \neq v_\ell$. Concernant BTD-MAC (i.e. BTD utilisant MAC pour résoudre chaque cluster), nous pouvons constater que la prochaine décision positive porte nécessairement sur une variable du cluster courant E_i, du moment où une variable de E_i ne figure pas dans une décision positive, et que seuls les domaines des variables futures présentes dans $Desc(E_i)$ sont susceptibles d'être impactés par le filtrage par AC (car $E_i \cap E_{p(i)}$ est un séparateur du graphe de contraintes et que toutes ses variables ont déjà été affectées).

Quand BTD a instancié de façon cohérente toutes les variables d'un cluster E_i, il tente de résoudre chaque sous-problème enraciné en un des clusters fils E_j de E_i. Plus précisément, pour un fils E_j et une suite de décisions courante Σ, il essaie de résoudre le sous-problème induit par les variables de $Desc(E_j)$ et l'ensemble de décisions $Pos(\Sigma)[E_i \cap E_j]$ (i.e. l'ensemble des décisions positives impliquant une variable de $E_i \cap E_j$). Une fois ce sous-problème résolu, en prouvant soit l'existence, soit l'inexistence de solution, il mémorise soit un good structurel, soit un nogood structurel. Formellement, étant donnés un cluster E_i et E_j un de ses fils, un *good structurel* (resp. *nogood structurel*) de E_i par rapport à E_j est une affectation cohérente A de $E_i \cap E_j$ telle que A peut (resp. ne peut pas) être étendue de façon cohérente sur $Desc(E_j)$ [8]. Dans le cas particulier de BTD-MAC, l'affectation cohérente A est représentée par la restriction de l'ensemble des décisions positives de Σ sur $E_i \cap E_j$, à savoir $Pos(\Sigma)[E_i \cap E_j]$. Ces (no)goods structurels peuvent être employés plus tard dans la recherche pour éviter d'explorer une partie redondante de l'arbre de recherche. En effet, dès que la suite de décisions courante Σ contient un good (resp. nogood) de E_i par rapport à E_j, BTD n'a plus besoin de résoudre à nouveau le sous-problème induit par $Desc(E_j)$ et $Pos(\Sigma)[E_i \cap E_j]$ car il a déjà établi que ce sous-problème possédait au moins une solution (resp. aucune). Dans le cas d'un good, il poursuit sa recherche avec le prochain cluster fils. Dans celui d'un nogood, il revient en arrière. Par exemple, considérons un CSP de 8 variables $x_1, \ldots, x_8$ ayant chacune pour domaine $\{a, b, c\}$ et dont le graphe de contraintes et une décomposition arborescente possible sont donnés à la figure 1. Supposons que la suite de décisions courante $\Sigma = \langle x_1 = a, x_2 \neq b, x_2 = c, x_3 = b \rangle$ a été construite selon un ordre sur les variables compatible avec l'ordre sur les clusters $E_1 < E_2 < E_3 < E_4$. BTD essaie de résoudre le sous-problème enraciné en E_2 et une fois résolu, il mémorise $\{x_2 = c, x_3 = b\}$ comme un good ou un nogood structurel de E_1 par rapport à E_2. Si, plus tard, BTD étudie la suite de décisions $\langle x_1 \neq a, x_3 = b, x_1 = b, x_2 \neq a, x_2 = c \rangle$ et que celle-ci est cohérente, il continuera sa recherche avec le prochain fils de E_1 (à savoir E_4), si $\{x_2 = c, x_3 = b\}$ a été mémorisé sous la forme d'un good, ou il revient en arrière à la dernière décision de E_1 si $\{x_2 = c, x_3 = b\}$ correspond à un nogood.

L'algorithme 1 correspond à l'algorithme BTD-MAC. Initialement, la suite de décisions courante et les ensembles N et G de (no)goods structurels mémorisés sont vides et la recherche débute avec les variables du cluster racine E_r. Etant donnés un cluster courant E_i et la suite de décisions courante Σ, les lignes 16-23 consistent à explorer le cluster E_i en affectant les variables de V_{E_i} (avec V_{E_i} l'ensemble des variables non-instanciées du cluster E_i) comme le ferait MAC tandis que les lignes 1-14 permettent de gérer les fils de E_i et donc d'utiliser et d'enregistrer des (no)goods struc-

2. On suppose que $E_i \cap E_{p(i)} = \emptyset$ si E_i est le cluster racine.

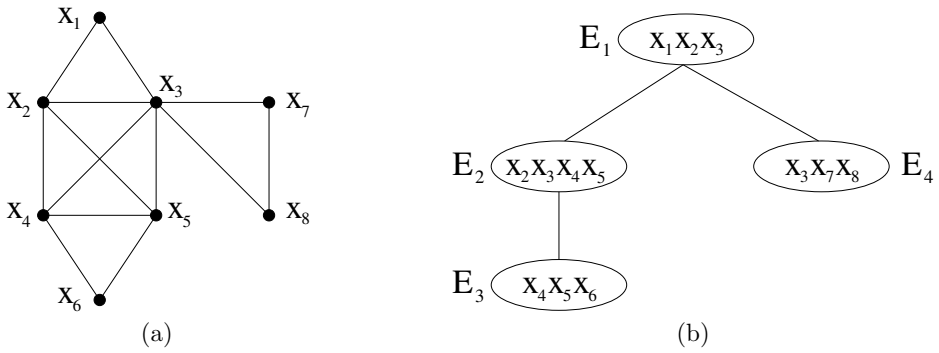


FIGURE 1 – Un graphe de contraintes de 8 variables (a) et une décomposition arborescente optimale (b).

Algorithme 1: BTD-MAC (**InOut** : $P = (X, D, C)$: CSP ; **In** : Σ : suite de décisions, E_i : cluster, V_{E_i} : ensemble de variables ; **InOut** : G : ensemble de goods, N : ensemble de nogoods)

```
1  if V_{E_i} = ∅ then
2      result ← true
3      S ← Sons(E_i)
4      while result and S ≠ ∅ do
5          Choose a cluster E_j ∈ S
6          S ← S\{E_j}
7          if Pos(Σ)[E_i ∩ E_j] is a nogood of E_i w.r.t. E_j in N
           then result ← false
8          else if Pos(Σ)[E_i ∩ E_j] is not a good of E_i w.r.t.
           E_j in G then
9              result ← BTD-MAC(P,Σ,E_j,E_j\(E_i ∩ E_j),G,N)
10             if result then
11                 Record Pos(Σ)[E_i ∩ E_j] as good of E_i w.r.t.
                   E_j in G
12             else
13                 Record Pos(Σ)[E_i ∩ E_j] as nogood of E_i
                   w.r.t. E_j in N
14     return result
15 else
16     Choose a variable x ∈ V_{E_i}
17     Choose a value v ∈ d_x
18     d_x ← d_x\{v}
19     if AC (P,Σ ∪ ⟨x = v⟩) ∧ BTD-MAC(P, Σ ∪ ⟨x = v⟩, E_i,
       V_{E_i}\{x}, G, N)= true then return true
20     else
21         if AC (P,Σ ∪ ⟨x ≠ v⟩) then
22             return BTD-MAC(P,Σ ∪ ⟨x ≠ v⟩,E_i,V_{E_i},G,N)
23         else return false
```

turels. BTD-MAC(P,Σ,E_i,V_{E_i},G,N) renvoie *true* s'il parvient à étendre Σ de façon cohérente sur les variables de $Desc(E_i)\backslash(E_i\backslash V_{E_i})$, *false* sinon. Sa complexité en temps est $O(n.s^2.e.\log(d).d^{w^+ + 2})$ pour une complexité en espace en $O(n.s.d^s)$ avec w^+ la largeur de la décomposition arborescente considérée et s la taille de la plus grande intersection entre deux clusters.

D'un point de vue pratique, généralement, BTD résout efficacement les CSP ayant une petite largeur arborescente [6, 7, 8]. Cependant, parfois, un mauvais choix de cluster racine peut dégrader significativement la qualité de la résolution. Le choix du cluster racine est crucial dans la mesure où il impacte directement l'ordre sur les variables, en particulier le choix des premières variables. Aussi, afin de faire un choix plus éclairé, nous avons sélectionné quelques instances de la compétition de solveurs de 2008 [3] pour lesquelles nous avons lancé BTD à partir de chaque cluster de la décomposition arborescente considérée. Nous avons d'abord constaté que, pour une même instance, les temps d'exécution varient de plusieurs ordres de grandeur selon le cluster racine choisi. Par exemple, pour l'instance scen11-f12 (qui est la plus facile de la famille scen11), BTD ne parvient à prouver l'absence de solution que pour 75 des 301 choix de racine possibles. Ensuite, nous avons observé que résoudre certains clusters (pas nécessairement le cluster racine) et leurs sous-problèmes se révèle être plus coûteux pour certains choix de racine que pour d'autres. Cela s'explique par le fait que le choix du cluster racine induit un ordre particulier sur les clusters et les variables. En particulier, comme pour un cluster E_i, BTD ne tient compte que des variables de $E_i\backslash(E_i \cap E_{p(i)})$, il ne manipule pas le même ensemble de variables pour E_i selon la racine choisie. Malheureusement, il semble utopique de vouloir proposer un choix de racine basé uniquement sur les propriétés de l'instance à résoudre car ce choix est intimement lié à l'efficacité de la résolution. Dans [7], une approche a été proposée pour choisir l'ordre sur les variables avec plus de liberté mais son efficacité demeure dépendante du choix de racine. Limiter l'impact du choix de racine est donc une nécessité. Dans la section 3, nous proposons une solution exploitant des techniques de redémarrage.

3 Exploitation des redémarrages au sein de BTD

Il est bien connu que n'importe quelle méthode exploitant des techniques de redémarrage doit autant que

3. Voir http ://www.cril.univ-artois.fr/CPAI08.

possible éviter d'explorer plusieurs fois les mêmes parties de l'espace de recherche et que la randomisation et l'apprentissage sont deux voies possibles pour atteindre ce but. Au niveau de l'apprentissage, BTD exploite déjà des (no)goods structurels. La première question qui se pose est de savoir s'il est possible de réutiliser des (no)goods structurels après un redémarrage et si oui, sous quelles conditions. De plus, suivant le moment où se produit le redémarrage, nous n'avons aucune garantie qu'un (no)good aura déjà été mémorisé. Aussi, une autre forme d'apprentissage est requise si on veut garantir une bonne efficacité pratique. Ici, nous considérons les nld-nogoods réduits (pour negative last decision nogoods) dont l'intérêt pratique a été mis en avant dans l'algorithme MAC+RST+NG [9]. Nous rappelons d'abord la notion de nogood dans le cas de MAC :

Définition 2 ([9]) *Etant donnés un CSP* $P = (X, D, C)$ *et un ensemble de décisions* Δ, $P_{|\Delta}$ *est le CSP* (X, D', C) *avec* $D' = (d'_{x_1}, \ldots, d'_{x_n})$ *tel que pour chaque décision positive* $x_i = v_i$, $d'_{x_i} = \{v_i\}$ *et pour chaque décision négative* $x_i \neq v_i$, $d'_{x_i} = d_{x_i} \backslash \{v_i\}$. *Pour le cas où* x_i *n'apparaît pas dans* Δ, *on a* $d'_{x_i} = d_{x_i}$. Δ *est un* nogood *de* P *si* $P_{|\Delta}$ *n'a pas de solution.*

Dans la suite, nous supposerons que pour toute variable x_i et toute valeur v_i, la décision positive $x_i = v_i$ est toujours considérée avant la décision $x_i \neq v_i$.

Proposition 1 ([9]) *Soit* $\Sigma = \langle \delta_1, \ldots, \delta_k \rangle$ *une suite de décisions prises le long d'une branche de l'arbre de recherche développé durant la résolution d'un CSP* P. *Pour toute sous-suite* $\Sigma' = \langle \delta_1, \ldots, \delta_\ell \rangle$ *préfixe de* Σ *telle que* δ_ℓ *est une décision négative, l'ensemble* $Pos(\Sigma') \cup \{\neg\delta_\ell\}$ *est un nogood (appelé un* nld-nogood réduit*) de* P *avec* $\neg\delta_\ell$ *la décision positive correspondant à* δ_ℓ.

Autrement dit, étant donnée une suite de décisions Σ prises le long d'une branche d'un arbre de recherche, chaque nld-nogood réduit caractérise la visite d'une partie inconsistante de cet arbre de recherche. Quand un redémarrage se produit, un algorithme comme MAC+RST+NG peut mémoriser de nouveaux nld-nogoods réduits et les exploiter ensuite pour éviter d'explorer à nouveau des parties déjà visitées de l'arbre de recherche. Il a été établi dans [9] que le calcul et l'utilisation de ces nld-nogoods réduits peuvent être faits efficacement, notamment en les stockant sous la forme d'une contrainte globale avec un propagateur spécifique pour établir la cohérence d'arc.

L'utilisation d'apprentissage au sein de BTD peut mettre en danger sa validité dès qu'on ajoute au problème initial une contrainte dont la portée n'est incluse dans aucun cluster. Par conséquent, la mémorisation des nld-nogoods réduits dans une contrainte globale portant sur toutes les variables, comme proposé dans [9], est impossible. Cependant, en exploitant les propriétés d'un ordre compatible sur les variables, la proposition 2 montre que cette contrainte globale peut être décomposée en une contrainte globale par cluster.

Proposition 2 *Soit* $\Sigma = \langle \delta_1, \ldots, \delta_k \rangle$ *une suite de décisions prises le long d'une branche de l'arbre de recherche développé durant la résolution d'un CSP* P *à l'aide d'une décomposition arborescente* (E, T) *et d'un ordre compatible sur les variables. Soit* $\Sigma[E_i]$ *la sous-suite construite en ne considérant que les décisions de* Σ *concernant des variables de* E_i. *Pour toute sous-suite* $\Sigma'_{E_i} = \langle \delta_{i_1}, \ldots, \delta_{i_\ell} \rangle$ *préfixe de* $\Sigma[E_i]$ *telle que* δ_{i_ℓ} *est une décision négative et que chaque variable de* $E_i \cap E_{p(i)}$ *apparaît dans une décision de* $Pos(\Sigma'_{E_i})$, *l'ensemble* $Pos(\Sigma'_{E_i}) \cup \{\neg\delta_{i_\ell}\}$ *est un nld-nogood réduit de* P.

Preuve : Soient P_{E_i} le sous-problème induit par les variables de $Desc(E_i)$ et Δ_{E_i} l'ensemble des décisions de $Pos(\Sigma_{E_i})$ relatives aux variables de $E_i \cap E_{p(i)}$. Comme $E_i \cap E_{p(i)}$ est un séparateur du graphe de contraintes, $P_{E_i|\Delta_{E_i}}$ est indépendant du reste du problème P. Considérons $\Sigma[E_i]$ la sous-suite de Σ qui ne contient que des décisions impliquant des variables de E_i. D'après la proposition 1 appliquée à $\Sigma[E_i]$ et $P_{E_i|\Delta_{E_i}}$, $Pos(\Sigma'_{E_i}) \cup \{\neg\delta_{i_\ell}\}$ est nécessairement un nld-nogood réduit. □

Il en découle les deux corollaires suivants qui permettent de majorer la taille des nogoods produits et de les comparer à ceux produits par la proposition 1.

Corollaire 1 *Etant donnée une décomposition arborescente de largeur* w^+, *les nogoods réduits produits par la proposition 2 sont de taille au plus* $w^+ + 1$.

Corollaire 2 *Sous les mêmes hypothèses que la proposition 2, pour chaque nld-nogood réduit* Δ *produit par la proposition 1, il existe au moins un nld-nogood réduit* Δ' *produit par la Proposition 2 tel que* $\Delta' \subseteq \Delta$.

En mémorisant des (no)goods structurels, BTD exploite déjà une forme particulière d'apprentissage. Tout (no)good structurel d'un cluster E_i par rapport à un cluster fils E_j est par définition orienté de E_i vers E_j. Cette orientation est induite directement par le choix d'un cluster racine. Quand un redémarrage se produit, BTD peut choisir un cluster différent comme cluster racine. Si c'est le cas, nous devons considérer des (no)goods structurels avec différentes orientations. La proposition 3 établit comment ces (no)goods structurels peuvent être utilisés de façon valide quand BTD exploite des techniques de redémarrage.

Proposition 3 *Un good structurel de E_i par rapport à E_j peut être utilisé si le choix courant de cluster racine induit que E_j est un fils de E_i.*
Un nogood structurel de E_i par rapport à E_j peut être utilisé quel que soit le choix de cluster racine.

Preuve : Soit un good Δ de E_i par rapport à E_j produit pour un cluster racine E_r. Par définition d'un good structurel, le sous-problème $P_{E_j|\Delta}$ possède au moins une solution et sa définition ne dépend que de Δ et du fait que E_j est un fils de E_i. Par conséquent, pour n'importe quel choix de cluster racine pour lequel E_j est un fils de E_i, Δ sera encore un good structurel de E_i par rapport à E_j et pourra donc être utilisé pour ne pas explorer des parties redondantes de l'espace de recherche.

Concernant les nogoods structurels, tout nogood structurel Δ de E_i par rapport à E_j est un nogood, et donc toute suite de décisions Σ telle que $\Delta \subseteq Pos(\Sigma)$ ne pourra être étendue en une solution, indépendamment du choix de cluster racine. Ainsi, les nogoods structurels sont utilisables quel que soit le choix de racine. □

Il s'ensuit que contrairement aux nogoods, pour les goods, l'orientation doit être prise en compte. Il serait donc plus pertinent des les appeler des *goods structurels orientés*.

L'algorithme 3 décrit l'algorithme BTD-MAC+RST qui exploite des techniques de redémarrage tout en mémorisant des nld-nogoods réduits et des (no)goods structurels. L'exploitation des techniques de redémarrage peut être vue comme le choix d'un cluster racine (ligne 3) et l'exécution d'une nouvelle instance de BTD-MAC+NG (ligne 4) à chaque redémarrage jusqu'à ce que le problème soit résolu en montrant l'(in)existence de solution. L'algorithme 2 présente l'algorithme BTD-MAC+NG. Comme BTD-MAC, étant donnés un cluster E_i et une suite de décisions courante Σ, BTD-MAC+NG explore le cluster E_i (lignes 16-27) en affectant les variables de V_{E_i} (avec V_{E_i} l'ensemble des variables non instanciées de E_i). Quand E_i est instancié de manière cohérente, il gère les fils de E_i et donc utilise et mémorise des (no)goods structurels (lignes 1-14). Les (no)goods structurels utilisés peuvent avoir été mémorisés durant l'appel courant à BTD-MAC+NG ou durant un appel précédent. En effet, si le premier appel à BTD-MAC+NG s'effectue avec des ensembles vides pour les ensembles G et N de goods et nogoods structurels, G et N ne sont pas réinitialisés à chaque redémarrage. Notons que leurs emplois (lignes 7-8) se font en accord avec la proposition 3. Ensuite, à la différence de BTD-MAC, BTD-MAC+NG peut stopper sa recherche aussitôt que la condition de redémarrage est atteinte (ligne 21). Si c'est le cas, il mémorise des nld-nogoods réduits par rapport à la suite de décisions Σ restreinte aux décisions impliquant les variables de E_i (ligne 22) en accord avec la proposition 2. La détection et la mémorisation des nld-nogoods réduits s'effectuent selon la méthode proposée dans [9] appliquée à la sous-suite de décisions de Σ n'impliquant que des décisions concernant des variables du cluster E_i. Nous considérons qu'une contrainte globale est associée à chaque cluster E_i pour manipuler les nld-nogoods enregistrés au niveau de E_i et que leur exploitation s'effectue au travers d'un propagateur spécifique quand la cohérence d'arc est appliquée (lignes 19 et 25) comme dans [9]. La condition de redémarrage peut porter sur des paramètres globaux (e.g. le nombre de retours en arrière accomplis depuis le début de l'appel courant à BTD-MAC+NG), des locaux (e.g. le nombre de retours en arrière accomplis dans le cluster courant ou le nombre de (no)goods structurels mémorisés) ou une combinaison des deux. BTD-MAC+NG(P,Σ,E_i,V_{E_i},G,N) renvoie :

- *true* s'il parvient à étendre Σ de façon cohérente sur les variables de $Desc(E_i)\backslash(E_i\backslash V_{E_i})$,
- $false$ s'il prouve que Σ ne peut pas s'étendre de façon cohérente sur les variables de $Desc(E_i)\backslash(E_i\backslash V_{E_i})$,
- *unknown* si un redémarrage se produit.

BTD-MAC+RST(P) renvoie *true* si P possède au moins une solution, $false$ sinon.

Théorème 1 *BTD-MAC+RST est correct, complet et termine.*

Preuve : BTD-MAC+NG diffère de BTD-MAC par l'exploitation des techniques de redémarrage, par l'enregistrement de nld-nogoods réduits et au niveau des ensembles G et N qui ne sont pas nécessairement vides au départ. Quand un redémarrage se produit, la recherche est arrêtée et des nld-nogoods réduits sont mémorisés de façon valide en accord avec la proposition 2. Concernant les (no)goods structurels, N et G ne contiennent que des (no)goods structurels valides et leurs usages (lignes 7-8) s'effectuent en accord avec la proposition 3. Par conséquent, comme BTD-MAC est correct et termine et que ces propriétés ne sont pas remises en cause par les différences entre BTD-MAC et BTD-MAC+NG, il en est de même pour BTD-MAC+NG. Concernant la complétude, comme BTD-MAC est complet, BTD-MAC+NG l'est aussi sous réserve qu'aucun redémarrage ne se produise. Par ailleurs, la survenue d'un redémarrage ne fait qu'interrompre la recherche. Elle ne remet donc pas en cause le fait que s'il avait eu une solution dans la partie de l'espace de recherche explorée par l'appel courant à BTD-MAC+NG, celui-ci aurait trouvé cette solution.

Algorithme 2: BTD-MAC+NG (**InOut** : $P = (X, D, C)$: CSP ; **In** : Σ : suite de décisions, E_i : cluster, V_{E_i} : ensemble de variables ; **InOut** : G : ensemble de goods, N : ensemble de nogoods)

```
1  if V_{E_i} = ∅ then
2      result ← true
3      S ← Sons(E_i)
4      while result = true and S ≠ ∅ do
5          Choose a cluster E_j ∈ S
6          S ← S\{E_j}
7          if Pos(Σ)[E_i ∩ E_j] is a nogood in N then
           result ← false
8          else if Pos(Σ)[E_i ∩ E_j] is not a good of E_i w.r.t.
           E_j in G then
9              result ←
               BTD-MAC+NG(P,Σ,E_j,E_j\(E_i ∩ E_j),G,N)
10             if result = true then
11                 Record Pos(Σ)[E_i ∩ E_j] as good of E_i w.r.t.
                   E_j in G
12             else if result = false then
13                 Record Pos(Σ)[E_i ∩ E_j] as nogood of E_i
                   w.r.t. E_j in N
14     return result
15 else
16     Choose a variable x ∈ V_{E_i}
17     Choose a value v ∈ d_x
18     d_x ← d_x\{v}
19     if AC (P,Σ ∪ ⟨x = v⟩) ∧ BTD-MAC+NG(P,
       Σ ∪ ⟨x = v⟩, E_i, V_{E_i}\{x}, G, N)= true then return
       true
20     else
21         if must restart then
22             Record nld-nogoods w.r.t. the decision sequence
               Σ[E_i]
23             return unknown
24         else
25             if AC (P,Σ ∪ ⟨x ≠ v⟩) then
26                 return
                   BTD-MAC+NG(P,Σ ∪ ⟨x ≠ v⟩,E_i,V_{E_i},G,N)
27             else return false
```

Comme BTD-MAC+RST ne fait que réaliser plusieurs appels à BTD-MAC+NG, il est forcément correct. Au niveau de la complétude, si l'appel à BTD-MAC+NG n'est pas interrompu par un redémarrage (ce qui est nécessairement le cas du dernier appel à BTD-MAC+NG si BTD-MAC+RST termine), la complétude de BTD-MAC+NG implique celle de BTD-MAC+RST. Par ailleurs, l'enregistrement de nld-nogoods réduits à chaque redémarrage garantit de ne pas explorer une partie de l'espace de recherche déjà explorée par un précédent appel à BTD-MAC+NG. Il s'ensuit qu'au fil des appels successifs à BTD-MAC+NG, on va explorer une partie de plus en plus réduite de l'espace de recherche. Ainsi, la terminaison et donc la complétude de BTD-MAC+RST sont garanties par l'enregistrement non limité de nld-nogoods effectué durant les différents appels à BTD-MAC+NG ainsi que par la terminaison et la complétude de BTD-MAC+NG. □

Algorithme 3: BTD-MAC+RST (**In** : $P = (X, D, C)$: CSP)

```
1  G ← ∅ ; N ← ∅
2  repeat
3      Choose a cluster E_r as root cluster
4      result ← BTD-MAC+NG (P,∅,E_r,E_r,G,N)
5  until result ≠ unknown
6  return result
```

Théorème 2 *BTD-MAC+RST a une complexité en temps en* $O(R.((n.s^2.e.\log(d) + w^+.N).d^{w^+ +2} + n.(w^+)^2.d))$ *et une complexité en espace en* $O(n.s.d^s + w^+.(d + N))$ *avec* w^+ *la largeur de la décomposition arborescente considérée, s la taille de la plus grande intersection* $E_i \cap E_j$*, R le nombre de redémarrages et N le nombre de nld-nogoods réduits mémorisés.*

Preuve : BTD-MAC sans nld-nogoods a une complexité en temps en $O(n.s^2.e.\log(d).d^{w^+ +2})$. D'après les propositions 4 et 5 de [9], mémoriser et gérer les nld-nogoods de taille au plus n peut s'effectuer respectivement en $O(n^2.d)$ et $O(n.N)$. Comme, d'après le corollaire 1, la taille des nld-nogoods est au plus $w^+ + 1$, ces deux opérations peuvent être accomplies respectivement en $O((w^+)^2.d)$ et $O(w^+.N)$. BTD-MAC+RST réalise au plus R appels à BTD-MAC. Par conséquent, on obtient une complexité en temps pour BTD-MAC+RST en $O(R.((n.s^2.e.\log(d) + w^+.N).d^{w^+ +2} + n.(w^+)^2.d))$.

En exploitant la structure de données proposée dans [9], la complexité en espace pour stocker les nld-nogoods réduits est en $O(w^+.(d+N))$ dans le pire des cas car, selon le corollaire 1, BTD-MAC+RST mémorise N nogoods de taille au plus $w^+ + 1$. Concernant la mémorisation des (no)goods structurels, BTD-MAC+RST possède la même complexité en espace que BTD, à savoir $O(n.s.d^s)$. Donc, au total, la complexité en espace est $O(n.s.d^s + w^+.(d+N))$. □

Si BTD-MAC+RST emploie une politique de redémarrage géométrique [16] basée sur le nombre de retours en arrière autorisés (i.e. un redémarrage se produit dès que le nombre de retours en arrière effectués dépasse le nombre de retours en arrière autorisés qui est initialement fixé à n_0 et augmenté d'un facteur multiplicatif r à chaque redémarrage), nous pouvons majorer le nombre de redémarrage :

Proposition 4 *Etant donnée une politique de redémarrage géométrique basée sur le nombre de retours en arrière avec un nombre de retours en arrière initialement fixé à* n_0 *et un facteur multiplicatif r, le nombre de redémarrage R est majoré par* $\left\lceil \frac{\log(n)+(w^+ +1).\log(d)-\log(n_0)}{\log(r)} \right\rceil$.

Preuve : Dans le pire des cas, le nombre de retours en arrière est majoré par $n.d^{w^+ +1}$ car on a au plus n

clusters et que le nombre de retours en arrière pour un cluster donné est au plus $O(d^{w^+ +1})$. Au $i$ème redémarrage, le nombre de retours en arrière autorisé est $n_0.r^i$. Dans le pire des cas, BTD-MAC+RST termine dès que $n_0.r^i \geq n.d^{w^+ +1}$, i.e. dès que $i \geq \frac{\log(n)+(w^+ +1).\log(d)-\log(n_0)}{\log(r)}$. □

4 Expérimentations

Dans cette section, nous évaluons l'intérêt pratique des redémarrages pour la résolution de CSP grâce à des méthodes par décomposition. Dans ce but, nous comparons BTD-MAC+RST avec BTD-MAC, MAC et MAC+RST+NG sur 647 instances (d'arité quelconque) issues de la compétition de solveurs de 2008. Les instances sélectionnées sont celles qui possèdent des décompositions arborescentes convenables (i.e. celles avec un rapport n/w^+ au moins égal à 2). Les décompositions arborescentes sont calculées avec l'algorithme Min-Fill [13] qui est considéré comme une des meilleures heuristiques de l'état de l'art [5]. Le temps d'exécution de BTD-MAC(+RST) inclut le temps de calcul de la décomposition arborescente. Tous les algorithmes exploitent l'heuristique de choix de variables *dom/wdeg* [2]. Pour le choix du cluster racine, nous avons testé plusieurs heuristiques. Nous présentons ici les deux meilleures :

- RW : on choisit le cluster qui maximise la somme des poids des contraintes dont la portée intersecte le cluster (les poids considérés sont ceux de dom/wdeg). Cette heuristique est aussi utilisée pour BTD-MAC.
- RA : on choisit alternativement soit le cluster qui contient la prochaine variable selon l'heuristique dom/wdeg appliquée à toutes les variables et qui maximise la somme des poids des contraintes dont la portée intersecte le cluster, soit le cluster suivant en classant les clusters dans l'ordre décroissant du rapport nombre de contraintes sur taille du cluster moins un.

Ces deux heuristiques visent à suivre le principe du first-fail. Le second cas de l'heuristique RA permet d'apporter de la diversité à la recherche. Les politiques de redémarrage utilisées reposent toutes sur le nombre de retours en arrière autorisés. Les valeurs présentées ici sont celles qui fournissent les meilleurs résultats parmi les valeurs testées. Plus précisément, pour MAC+RST+NG, nous exploitons une politique géométrique pour laquelle le nombre initial de retours en arrière autorisés est 100 et le facteur multiplicatif 1.1. BTD-MAC+RST avec RW utilise une politique géométrique avec un facteur de 1.1 et initialement 50 retours en arrière sont autorisés. Pour RA, nous appliquons une politique géométrique avec un facteur 1.1

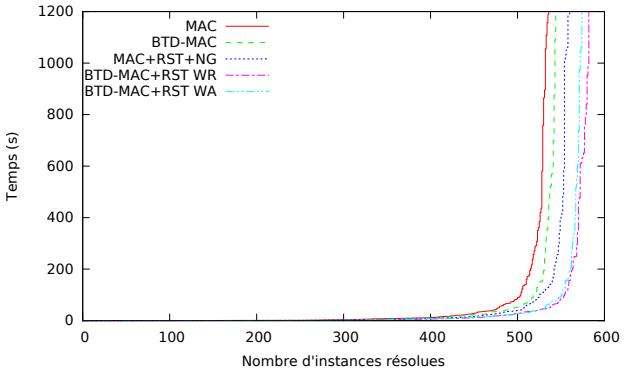


FIGURE 2 – Nombre total d'instances résolues par chaque algorithme.

et un nombre initial de retours en arrière autorisé de 75 quand le cluster est choisi selon la première règle. Dans le cas de la seconde, nous utilisons un nombre constant de retours en arrière autorisés fixé également à 75. Les expérimentations sont effectuées avec nos propres solveurs implémentés en C++, sur un PC basé sur Linux avec un processeur Intel Pentium IV cadencé à 3,2 GHz et 1 Go de mémoire. Le temps d'exécution est limité à 1 200 s (sauf pour la table 1).

La figure 2 présente le nombre total d'instances résolues par chacun des algorithmes considérés. D'abord, nous pouvons noter que, pour BTD-MAC+RST, les deux heuristiques RW et RA se comportent de manière similaire. Ensuite, il apparaît clairement que BTD-MAC+RST résout plus d'instances que les autres algorithmes. Par exemple, BTD-MAC+RST résout 582 instances en 15 863 s avec RW (resp. 574 instances en 13 280 s avec RA) alors que MAC+RST+NG n'en résout que 560 en 16 943 s. Sans les techniques de redémarrage, le nombre d'instances résolues est encore plus faible avec 536 instances en 18 063 s pour MAC et 544 instances en 13 256 s pour BTD-MAC.

Afin de mieux analyser le comportement des différents algorithmes, nous considérons maintenant les résultats obtenus par famille d'instances [4]. La table 2 fournit le nombre d'instances résolues et la somme des temps d'exécution pour ces instances pour chacun des algorithmes considérés tandis que la table 3 présente la somme des temps d'exécution en ne considérant que les instances résolues par tous les algorithmes. Nous pouvons d'abord constater, que pour certains types d'instances, comme les instances de la famille *graph coloring*, l'utilisation de techniques de redémar-

4. Notons que nous ne prenons pas en compte toutes les instances d'une famille, mais seulement celles qui ont une décomposition arborescente adéquate (cf. $n/w^+ \geq 2$).

rage ne permet pas d'améliorer l'efficacité de BTD-MAC+RST par rapport à MAC+RST+NG ou BTD-MAC. Par contre, pour les autres familles considérées, on peut observer que BTD-MAC+RST obtient des résultats intéressants. Ces bons résultats sont parfois dus à la décomposition arborescente (e.g. pour les familles *dubois* ou *haystacks*) car ils sont proches de ceux de BTD-MAC. De même, dans certains cas, ils résultent principalement de l'emploi de techniques de redémarrage (e.g. pour les familles *jobshop* ou *geom*) et ils sont alors voisins de ceux de MAC+RST+NG. Enfin, dans d'autres cas, BTD-MAC+RST tire pleinement partie à la fois de la décomposition arborescente et des techniques de redémarrage (e.g. pour les familles *renault*, *superjobshop* ou *scen11*). Dans de tels cas, il surclasse clairement les trois autres algorithmes. Par exemple, il est deux fois plus rapide que MAC+RST+NG pour résoudre les instances de la famille *scen11*, qui contient les instances d'allocation de fréquence les plus difficiles [3]. La table 1 présente les temps d'exécution de MAC+RST+NG et de BTD-MAC+RST pour ces instances. Nous pouvons remarquer que BTD-MAC ne résout que les trois plus simples. Cela s'explique par un mauvais choix de cluster racine. Il s'avère que, pour toutes les instances de cette famille, la plupart des choix de cluster racine conduisent à passer beaucoup de temps à résoudre certains sous-problèmes. Aussi, les techniques de redémarrages se révèlent ici très utiles.

Enfin, nous avons observé que BTD-MAC+RST est généralement plus efficace sur les instances ne possédant pas de solution que MAC+RST. Par exemple, si on considère les instances dépourvues de solution qui sont résolues par tous les algorithmes, il nécessite 4 260 s pour les résoudre contre 7 105 s pour MAC+RST. Un tel phénomène s'explique en partie par l'utilisation des décompositions arborescentes. En effet, si BTD-MAC+RST explore, au début de la recherche, un cluster sans solution, il peut rapidement conclure que le problème entier n'en a pas non plus.

Instance	MAC+RST+NG	BTD-MAC+RST
scen11-f12	0,51	0,30
scen11-f11	0,50	0,30
scen11-f10	0,65	0,35
scen11-f9	1,32	1,54
scen11-f8	1,60	1,78
scen11-f7	12,93	6,81
scen11-f6	20,23	9,86
scen11-f5	102	45,72
scen11-f4	397	202
scen11-f3	1 277	609
scen11-f2	3 813	1 911
scen11-f1	9 937	5 014

TABLE 1 – Temps de résolution en s (sans limite) pour l'instance scen11.

5 Conclusion

Dans cet article, nous avons d'abord décrit comment intégrer MAC dans BTD. Nous avons ensuite montré comment il est possible d'améliorer les méthodes de résolution exploitant des décompositions en leur ajoutant le concept de redémarrage. Cela nous a conduit à proposer une version étendue de BTD, à savoir la méthode BTD-MAC+RST. Pour cela, nous avons d'abord décrit comment les nogoods classiques peuvent être incorporés dans une méthode de résolution par décomposition tout en préservant la structure induite par la décomposition considérée. Ensuite, nous avons introduit la notion de *good structurel orienté*. En effet, si les nogoods structurels peuvent être directement utilisés quand BTD effectue des redémarrages, les goods doivent vérifier certaines propriétés liées à l'ordre d'exploration d'une décomposition arborescente, d'où la nécessité de les orienter. Dans la dernière partie du papier, les expérimentations ont clairement démontré l'intérêt pratique de l'utilisation des redémarrages dans les méthodes de résolution par décomposition. Cela permet, dans les faits, de dépasser le problème induit par l'ordre d'exploration des clusters, qui, très souvent, nuit de façon significative à l'efficacité pratique. Ces résultats ont également prouvé que l'ajout des redémarrages au sein de BTD permet de surclasser significativement les algorithmes MAC et MAC+RST+NG quand la topologie du réseau de contraintes possède une largeur convenable.

Comme perspectives de ce travail, nous pensons d'abord que des améliorations sont possibles au niveau des politiques de redémarrage qui peuvent être vues sous un nouvel éclairage. Il serait particulièrement intéressant de définir de nouvelles politiques spécifiques au cas des décompositions en considérant des politiques locales ou des combinaisons de politiques locales et globales mais aussi, d'adapter d'autres stratégies comme les redémarrages de Luby [10]. De plus, nous pouvons envisager des choix plus éclairés pour le choix du cluster racine en exploitant des informations plus riches et spécifiques aux décompositions, comme par exemple, le nombre de (no)goods dans un cluster. Enfin, cette approche pourrait être appliquée à un niveau supérieur. Aujourd'hui, nous exploitons des redémarrages pour contourner le problème du choix du cluster racine. Toutefois, l'efficacité des méthodes de résolution par décomposition est également fortement liée à la qualité de la décomposition employée. Aussi, pour contourner le problème du choix d'une décomposition convenable, nous pourrions envisager d'utiliser une nouvelle décomposition à chaque redémarrage. Tout en offrant plus de liberté, cette alter-

Famille	#inst.	MAC		BTD-MAC		MAC+RST+NG		BTD-MAC+RST			
								RW		RA	
		#rés.	temps	#rés.	temps	#rés.	temps	#rés.	temps	#rés.	temps
dubois	13	5	2 232	13	0,03	5	2 275	13	0,04	13	0,05
geom	83	83	415	83	819	83	479	83	468	83	460
graphColoring	39	29	1 989	33	1 291	29	2 783	34	2 825	33	2 769
haystacks	46	2	5,82	8	169	2	4,43	8	172	8	172
jobshop	46	37	617	35	469	46	14,87	46	13,15	46	10,93
renault	50	50	23,89	50	86,81	50	24,30	50	22,96	50	24,73
pret	8	4	250	8	0,05	4	552	8	0,06	8	0,05
scens11	12	8	1 632	3	1,25	9	537	10	878	10	882
Super-jobShop	46	19	1 648	21	1 179	33	2 315	34	1 553	27	449
travellingSalesman-20	15	15	191	15	229	15	214	15	346	15	294

TABLE 2 – Nombre total d'instances résolues et temps total de résolution en s par famille pour chaque algorithme.

Famille	#inst.	MAC	BTD-MAC	MAC+RST+NG	BTD-MAC+RST	
					RW	RA
dubois	5 / 13	2 232	0,01	2 275	0,01	0,01
graphColoring	27 / 39	951	1 051	1 308	846	1 277
haystacks	2 / 46	5,82	0	4,43	0	0,01
jobshop	33 / 46	392	468	5,63	5,10	4,48
pret	4 / 8	250	0,01	552	0,02	0
rlfapScens11	3 / 12	2,75	1,25	1,66	0,95	1,10
Super-jobShop	16 / 46	1 275	830	14,83	9,60	16,04

TABLE 3 – Temps de résolution en s pour chaque algorithme pour les instances résolues par tous les algorithmes.

native engendre des questions plus complexes car les nogoods, structurels ou non, et les goods ne seraient utilisables qu'en respectant des conditions encore plus restrictives.

Références

[1] C. Berge. *Graphs and Hypergraphs.* Elsevier, 1973.

[2] F. Boussemart, F. Hemery, C. Lecoutre, and L. Saïs. Boosting systematic search by weighting constraints. In *ECAI*, pages 146–150, 2004.

[3] C. Cabon, S. de Givry, L. Lobjois, T. Schiex, and J. P. Warners. Radio Link Frequency Assignment. *Constraints*, 4 :79–89, 1999.

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

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

[6] P. Jégou, S.N. Ndiaye, and C. Terrioux. 'Dynamic Heuristics for Backtrack Search on Tree-Decomposition of CSPs. In *IJCAI*, pages 112–117, 2007.

[7] P. Jégou, S.N. Ndiaye, and C. Terrioux. Dynamic Management of Heuristics for Solving Structured CSPs. In *CP*, pages 364–378, 2007.

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

[9] C. Lecoutre, L. Saïs, S. Tabary, and V. Vidal. Recording and Minimizing Nogoods from Restarts. *JSAT*, 1(3-4) :147–167, 2007.

[10] M. Luby, A. Sinclair, and D. Zuckerman. Optimal speedup of Las Vegas algorithms. *Inf. Process. Lett.*, 47(4) :173–180, 1993.

[11] J. Petke. *On the bridge between Constraint Satisfaction and Boolean Satisfiability.* PhD thesis, University of Oxford, 2012.

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

[13] D. J. Rose. A graph theoretic study of the numerical solution of sparse positive denite systems of linear equations. In *Graph Theory and Computing*, pages 183–217. R.C. Read (ed.), Academic Press, New York, 1973.

[14] F. Rossi, P. van Beek, and T. Walsh. *Handbook of Constraint Programming.* Elsevier, 2006.

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

[16] T. Walsh. Search in a small world. In *IJCAI*, pages 1172–1177, 1999.

Top-k SAT et son application à la fouille de données

Said Jabbour Lakhdar Sais Yakoub Salhi

CRIL - CNRS, Université d'Artois, France
F-62307 Lens Cedex, France
{jabbour, sais, salhi}@cril.fr

Résumé

Dans cet article, nous introduisons un nouveau problème, appelé Top-k SAT, qui consiste à énumérer les modèles top-k d'une formule propositionnelle. Un modèle top-k est défini comme un modèle ayant moins de k modèles qui lui sont préférés par rapport à une relation de préférence. Nous démontrons que Top-k SAT généralise deux problèmes connus : le problème Partial Max-SAT et le problème des modèles minimaux. En outre, nous proposons un algorithme général pour Top-k SAT. Ensuite, nous donnons la première application de notre cadre déclaratif en fouille de données, à savoir le problème de l'énumération des motifs ensemblistes top-k fréquents ayant des tailles supérieures ou égales à min ($\mathcal{FCIM}^k_{min}$). Enfin, pour mettre en évidence les aspects déclaratifs intéressants de notre cadre, nous fournissons des encodages de nombreuses variantes de $\mathcal{FCIM}^k_{min}$ dans le problème Top-k SAT.

Abstract

In this paper, we introduce a new problem, called Top-k SAT, that consists in enumerating the Top-k models of a propositional formula. A Top-k model is defined as a model with less than k models preferred to it with respect to a preference relation. We show that Top-k SAT generalizes two well-known problems : the partial Max-SAT problem and the problem of computing minimal models. Moreover, we propose a general algorithm for Top-k SAT. Then, we give the first application of our declarative framework in data mining, namely, the problem of enumerating the Top-k frequent closed itemsets of length at least min ($\mathcal{FCIM}^k_{min}$). Finally, to show the nice declarative aspects of our framework, we encode several other variants of $\mathcal{FCIM}^k_{min}$ into the Top-k SAT problem.

1 Introduction

Le problème de l'extraction de motifs ensemblistes fréquents est fondamental en fouille et analyse de données. Il possède des applications dans de nombreux domaines, en particulier, dans l'analyse des données volumineuses. Depuis le premier article d'Agrawal [1] sur les règles d'association et l'extraction de motifs ensemblistes, un nombre important de travaux, défis et projets ont vu le jour montrant l'intérêt réel de ce problème [21].

Des progrès importants ont été réalisés en fouille de données en termes d'implémentations, de plates-formes, de bibliothèques, etc. Comme indiqué dans [21], plusieurs travaux traitent de la conception d'algorithmes d'exploration de données hautement évolutives pour les données à grande échelle. Un problème important en extraction de motifs ensemblistes et en fouille de données de manière général, concerne la grande taille de la sortie à partir de laquelle il est difficile pour l'utilisateur d'extraire des informations pertinentes. Par conséquent, il est primordial de réduire la taille de la sortie en exploitant la structure des données objets de l'extraction. Par exemple, le calcul des motifs fermés, le calcul des motifs maximaux et le calcul d'autre formes condensées font partie des techniques connues permettant de réduire la taille de la sortie.

La majeure partie des travaux au sujet de l'extraction de motifs ensemblistes fréquents passe par l'utilisation d'un quorum λ. En un sens, il permet à l'utilisateur de contrôler dans une certaine mesure la taille de la sortie en extrayant uniquement les motifs ensemblistes couvrant au moins λ transactions. Cependant, il est difficile pour un utilisateur de fournir un seuil approprié. Comme mentionné dans [9], un seuil petit

peut mener à produire un nombre important de motifs ensemblistes, alors qu'un seuil grand peut mener à une sortie vide. Dans [9], les auteurs propose d'extraire les n motifs les plus intéressants possédant des tailles arbitraires en utilisant un ordre total sur les motifs. Dans [12], il est proposé d'extraire les motifs top-k fréquents et fermés ayant des tailles supérieures à une borne donnée min où k est le nombre désiré de motifs à extraire. Les auteurs démontrent que fixer une borne inférieure pour les tailles des motifs à extraire est beaucoup plus facile que de fixer un seuil pour les fréquences. Depuis l'introduction du problème de l'extraction de motifs top-k, plusieurs travaux de recherche ont exploré son utilisation en fouille de graphes (e.g. [13, 23]) et en d'autre tâches en fouille de données (e.g. [14, 15]). Ce cadre peut être considéré comme un bon moyen permettant l'extraction des k motifs préférés suivant des mesures spécifiques. Partant de ce constat, notre objectif dans cet article est de définir un cadre général fondée sur la logique propositionnelle pour l'extraction des motifs préférés top-k selon une relation de préférence prédéfinie.

La notion de préférence joue un rôle central dans plusieurs disciplines comme l'économie, la recherche opérationnelle et la théorie de décision de manière générale. Elle est importante pour la conception de systèmes intelligents qui prennes en charge des décisions. La modélisation et le raisonnement avec des préférences prennent une place importante en intelligence artificielle et les domaines qui lui sont liés comme le raisonnement non-monotone, la planification, le diagnostique, etc. Par exemple, l'introduction de la sémantique préférentielle pour le raisonnement non-monotone par Shoham [19] a permis de fournir un cadre unificateur où la logique non-monotone est réduite à une logique standard avec une relation de préférences sur ses modèles. Il est important de noter que plusieurs modèles pour la représentation et le raisonnement sur les préférences ont été proposés. Les contraintes souple [16], par exemple, sont une des façons les plus générales de traiter les préférences quantitatives, alors que l'approche CP-net (Conditional Preferences networks) [3] est plus appropriée dans le cas qualitatif. Il existe une littérature importante au sujet des préférences, nous renvoyons le lecteurs intéressé aux articles [24, 4, 7] pour une vue d'ensemble.

Dans cet article, nous nous concentrons sur les préférences qualitatives définies par une relation de préférences sur les modèles d'une formule propositionnelle. L'étude des préférences en logique propositionnelle n'a pas reçu beaucoup d'attention. Il y a néanmoins des travaux importants comme dans [18] où on trouve une approche pour la résolution du problème de la cohérence en présence de préférences qualitatives sur les littéraux. Les auteurs démontre en particulier que la procédure DPLL peut être facilement adaptée pour le calcul de modèles optimaux selon une relation d'ordre partielle. La problématique du calcul des modèles optimaux en utilisant DPLL a également été étudiée dans [6].

La contribution de cet article est double. Tout d'abord, nous proposons un cadre générique pour le traitement des préférences qualitatives au sein du problème de la cohérence propositionnelle. Nous considérons des préférences qualitatives définies en utilisant une relation réflexive et transitive sur les modèles d'une formule propositionnelle. Cela est réalisé par l'introduction d'un nouveau problème, nommé Top-k SAT, défini comme le problème d'énumération des modèles top-k d'une formules propositionnelle. Un modèle top-k est défini comme un modèle ayant moins de k modèles qui lui sont préférés par rapport à une relation de préférence. Nous démontrons que Top-k SAT généralise deux problèmes connus, à savoir *Partial Max-SAT* et le problème d'énumération des modèles minimaux. Nous définissons en outre un type particulier de relations de préférences qui nous permet d'introduire un algorithme générique pour le calcul des modèles top-k.

Quant à notre deuxième contribution, nous introduisons la première application de notre cadre déclaratif en fouille de données. Plus précisément, nous considérons le problème de l'extraction des motifs ensemblistes fermés top-k [25]. Dans ce problème, le seuil de fréquence n'est pas connu. Dans le problème de l'extraction de motifs ensemblistes fréquents, la notion de motif top-k est introduite comme une alternative à l'utilisation d'un seuil de fréquence. Il s'agit d'une manière élégante de réduire la taille de la sortie. Dans cette article, nous fournissons une encodage dans Top-k SAT du problème de l'extraction des motifs ensemblistes fermés top-k. Pour montrer les bons aspects déclaratifs de notre cadre, nous fournissons également d'autres encodages de variantes de ce problème de fouille de données. Enfin, l'étude expérimentale préliminaire sur certains jeux de données montre la faisabilité de notre approche.

2 Définitions préliminaires et notations

Dans cette section, nous décrivons le problème de cohérence d'une formule propositionnelle, appelé SAT. Nous considérons les formules propositionnelles en forme normale conjonctive (CNF). Une formule CNF Φ correspond à une conjonction de *clauses*, où une clause est une disjonction de *littéraux*. Un littéral est soit positif (p) ce qui correspond à une variable propositionnelle, soit négatif ($\neg p$) dans le sens où il est la né-

gation d'une variable propositionnelle. Les deux littéraux p et $\neg p$ sont considérés comme *complémentaires*. Une formule CNF peut également être vue comme un ensemble de clauses, et une clause comme un ensemble de littéraux. Nous rappelons que toute formule propositionnelle peut être transformée en une formule CNF en utilisant la méthode de Tseitin [22]. Nous utilisons la notation $Var(\Phi)$ pour représenter l'ensemble de variables apparaissant dans Φ.

Une *interprétation* $\mathcal{M}$ d'une formule propositionnelle Φ est une fonction qui associe une valeur $\mathcal{M}(p) \in \{0, 1\}$ (0 pour *faux* et 1 pour vrai) aux variables propositionnelles $p \in V$ telle que $Var(\Phi) \subseteq V$. Elle est étendue aux formule de la manière usuelle. Un *modèle* d'une formule Φ est une interprétation $\mathcal{M}$ qui vérifie $\mathcal{M}(\Phi) = 1$. Le problème SAT consiste à décider si une formule CNF possède un modèle ou non.

Nous utilisons la notation $\bar{l}$ afin de faire référence au littéral complémentaire de l. Plus précisément, si $l = p$ alors $\bar{l}$ est $\neg p$ et si $l = \neg p$ alors $\bar{l}$ est p. Pour tout ensemble de littéraux L, l'ensemble $\bar{L}$ est défini comme $\{\bar{l} \mid l \in L\}$. En outre, nous utilisons la notation $\overline{\mathcal{M}}$ ($\mathcal{M}$ est une interprétation définie sur $Var(\Phi)$) pour représenter la clause $\bigvee_{p \in Var(\Phi)} s(p)$, où $s(p) = p$ si $\mathcal{M}(p) = 0$, $\neg p$ sinon.

Soient Φ une formule CNF et $\mathcal{M}$ une interprétation définie sur $Var(\Phi)$. Nous utilisons la notation $\mathcal{M}(\Phi)$ pour représenter l'ensemble de clause ayant comme valeur de vérité 1 en utilisant $\mathcal{M}$. Considérons maintenant un ensemble de variables propositionnelles tel que $X \subseteq Var(\Phi)$. La notation $\mathcal{M} \cap X$ est utilisée pour représenter l'ensemble de variables $\{p \in X | \mathcal{M}(p) = 1\}$, tandis que $\mathcal{M}_{|X}$ est utilisée pour représenter la restriction de $\mathcal{M}$ à X.

3 Préférences et modèles top-k

Soient Φ une formule propositionnelle et Λ_Φ l'ensemble de ses modèles. Une *relation de préférences* $\succeq$ définies sur Λ_Φ est une relation binaire réflexive est transitive. L'énoncé $\mathcal{M} \succeq \mathcal{M}'$ signifie que $\mathcal{M}$ est au moins aussi préféré que $\mathcal{M}'$. La notation $P(\Phi, \mathcal{M}, \succeq)$ est utilisée pour représenter le sous-ensemble de Λ_Φ défini comme suit :

$$P(\Phi, \mathcal{M}, \succeq) = \{\mathcal{M}' \in \Lambda_\Phi \mid \mathcal{M}' \succ \mathcal{M}\}$$

où $\mathcal{M}' \succ \mathcal{M}$ correspond au deux énoncés $\mathcal{M}' \succeq \mathcal{M}$ et $\mathcal{M} \not\succeq \mathcal{M}'$. En d'autres termes, il correspond à l'ensemble des modèles préférés à $\mathcal{M}$.

Nous introduisons maintenant une relation d'équivalence $\approx_X$ définie sur $P(\Phi, \mathcal{M}, \succeq)$, où X est un ensemble de variables. Elle est définie comme suit :

$$\mathcal{M}' \approx_X \mathcal{M}'' \text{ iff } \mathcal{M}' \cap X = \mathcal{M}'' \cap X$$

Ainsi, l'ensemble $P(\Phi, \mathcal{M}, \succeq)$ peut être décomposé en un ensemble de classes d'équivalence par $\approx_X$, noté $[P(\Phi, \mathcal{M}, \succeq)]^X$. Dans notre contexte, cette relation d'équivalence est utilisée afin de prendre en compte uniquement un sous-ensemble de variables propositionnelles. Par exemple, la transformation de Tseitin introduit de nouvelles variables qui peuvent n'avoir aucune importance dans le cas de certaines relations de préférences.

Définition 1 (Modèle top-k) *Soient Φ un formule propositionnelle, $\mathcal{M}$ un modèle de Φ, $\succeq$ une relation de préférences définie sur les modèles de Φ, X un ensemble de variables propositionnelles et k un entier naturel supérieur ou égal à 1. $\mathcal{M}$ est un modèle top-k selon $\succeq$ et X ssi $|[P(\Phi, \mathcal{M}, \succeq)]^X| \leq k - 1$.*

Il est important de noter que l'ensemble des modèles top-k n'est pas nécessairement égal à k. En effet, il peut être strictement plus grand ou plus petit que k. Par exemple, une formule incohérente possède 0 modèle top-k. Par ailleurs, si la relation de préférences correspond à un ordre total alors le nombre des modèles top-k est nécessairement inférieur ou égal à k.

On peut aisément constater que la propriété de monotonie suivante est satisfaite : si $\mathcal{M}$ est un modèle top-k et $\mathcal{M}' \succeq \mathcal{M}$, alors $\mathcal{M}'$ est aussi un modèle top-k.

Le problème Top-k SAT. Soient Φ une formule propositionnelle, $\succeq$ une relation de préférences définie sur l'ensemble des modèles de Φ, X un ensemble de variables propositionnelles et k un entier naturel supérieur ou égal à 1. Le problème Top-k SAT consiste à calculer un ensemble $\mathcal{L}$ des modèles top-k de Φ selon $\succeq$ et X vérifiant les deux propriétés suivantes :

1. pour tout modèle top-k $\mathcal{M}$, il existe $\mathcal{M}' \in \mathcal{L}$ tel que $\mathcal{M} \approx_X \mathcal{M}'$; et
2. pour tous $\mathcal{M}$ et $\mathcal{M}'$ dans $\mathcal{L}$, si $\mathcal{M} \neq \mathcal{M}'$ alors $\mathcal{M} \not\approx_X \mathcal{M}'$.

Les deux précédentes propriétés viennent du fait que l'on est uniquement intéressé par les valeurs de vérité des variables dans X. En effet, la première propriété signifie que, pour tout modèle top-k, il existe un modèle dans $\mathcal{L}$ qui lui est équivalent selon $\approx_X$. De plus, la second propriété signifie que $\mathcal{L}$ ne peut contenir deux modèles top-k équivalents.

Dans la prochaine définition, nous introduisons un type particulier de relations de préférences, appelées relation de δ-préférences, qui nous permet d'introduire un algorithme générique pour le calcul des modèles top-k.

Définition 2 *Soient Φ une formule propositionnelle et $\succeq$ une relation de préférences définie sur les modèles de Φ. La relation $\succeq$ est une relation de relation*

de δ-préférences, si il existe une fonction polynomiale en temps $f_{\succeq}$ de l'ensemble des interprétations à l'ensemble des formules CNF telle que, pour tout modèle $\mathcal{M}$ de Φ et pour toute interprétation $\mathcal{M}'$, $\mathcal{M}'$ est un modèle de $\Phi \wedge f_{\succeq}(\mathcal{M})$ ssi $\mathcal{M}'$ est un modèle de Φ et $\mathcal{M} \not\succ \mathcal{M}'$.

Notons que, pour un modèle donné $\mathcal{M}$ d'une formule Φ, $f_{\succeq}(\mathcal{M})$ est une formule telle que quand on l'ajoute à Φ avec $\overline{\mathcal{M}}$, les modèles de la formule résultante sont différents de $\mathcal{M}$ et ils sont au moins aussi préférés que $\mathcal{M}$. Intuitivement, cela peut être vu comme un moyen d'introduire des bornes inférieures durant le processus d'énumération. Dans la suite de l'article, nous considérons uniquement les relations de δ-préférences.

3.1 Top-k SAT et Partial MAX-SAT

Dans cette section, nous montrons que le problème Top-k SAT généralise le problème *Partial MAX-SAT* [10]. Dans ce problème, chaque clause est soit souple, soit dure, l'objectif étant de trouver une interprétation qui satisfait toutes les clauses dures avec un nombre maximum de clauses souples. Le problème MAX-SAT est un cas particulier du problème Partial MAX-SAT où toutes les clauses sont souples.

Soit $\Phi = \Phi_h \wedge \Phi_s$ une instance du problème Partial MAX-SAT telle que Φ_h est sa partie dure et Φ_s est sa partie souple. La relation notée $\succeq_{\Phi_s}$ est une relation de préférences définie comme suit : pour tous modèles $\mathcal{M}$ et $\mathcal{M}'$ de Φ_h définis sur $Var(\Phi_h \wedge \Phi_s)$, $\mathcal{M} \succeq_{\Phi_s} \mathcal{M}'$ si et seulement si $|\mathcal{M}(\Phi_s)| \geq |\mathcal{M}'(\Phi_s)|$.

Il est à noter que $\succeq_{\Phi_s}$ est une relation de δ-préférences. En effet, la fonction $f_{\succeq_{\Phi_s}}$ peut être définie comme suit :

$$f_{\succeq_{\Phi_s}}(\mathcal{M}) = (\bigwedge_{C \in \Phi_s} p_C \leftrightarrow C) \wedge \sum_{C \in \Phi_s} p_C \geq |\mathcal{M}(\Phi_s)|$$

où p_C pour $C \in \Phi_s$ sont des variables propositionnelles nouvelles.

Les modèles top-1 de Φ_h selon $\succeq_{\Phi_s}$ et $Var(\Phi)$ correspondent à l'ensemble de toutes les solutions de Φ dans Partial MAX-SAT. Naturellement, ils constituent les modèles les plus préférés suivant $\succeq_{\Phi_s}$, et cela signifie qu'ils satisfont Φ_h et le nombre maximum de clauses dans Φ_s. Ainsi, le problème Top-k SAT peut être vu comme une généralisation de celui de Partial MAX-SAT.

La formule $f_{\succeq_{\Phi_s}}(\mathcal{M})$ implique une contrainte connue sous le nom de contrainte de cardinalité (inégalité linéaire 0/1). De nombreux encodages polynomiaux en formule CNF de cette contrainte existent dans la littérature. Le premier encodage a été proposé par Warners dans [26]. Récemment, d'autres encodages efficaces ont été fournis, la majeure partie améliore l'efficacité de la propagation de contrainte (e.g. [2, 20]).

3.2 Top-k SAT et modèles X-minimaux

Soient $\mathcal{M}$ et $\mathcal{M}'$ deux interprétations et X un ensemble de variables propositionnelles. Le modèle $\mathcal{M}$ est dit plus petit que $\mathcal{M}'$ selon X, écrit $\mathcal{M} \preceq_X \mathcal{M}'$, si $\mathcal{M} \cap X \subseteq \mathcal{M}' \cap X$. Nous considérons maintenant $\preceq_X$ comme une relation de préférences, i.e., $\mathcal{M} \preceq_X \mathcal{M}'$ signifie que $\mathcal{M}$ est au moins aussi préféré que $\mathcal{M}'$.

Nous démontrons maintenant que $\preceq_X$ est une relation de δ-préférences. Dans ce cadre, nous définissons $f_{\preceq_X}$ comme suit :

$$f_{\preceq_X}(\mathcal{M}) = (\bigvee_{p \in \mathcal{M} \cap X} \overline{p}) \vee \bigwedge_{p' \in X \setminus \mathcal{M}} \overline{p'}$$

Effectivement, $\mathcal{M}'$ est un modèle de la formule $\Phi \wedge \overline{\mathcal{M}} \wedge f_{\preceq_X}(\mathcal{M})$ si et seulement si $\mathcal{M}'$ est un modèle Φ, $\mathcal{M}' \neq \mathcal{M}$, et soit $\mathcal{M}' \cap X = \mathcal{M} \cap X$ soit $(\mathcal{M} \cap X) \setminus (\mathcal{M}' \cap X) \neq \emptyset$. Les deux dernières propriétés signifient que $\mathcal{M} \not\prec_X \mathcal{M}'$. En fait, si $\mathcal{M}'$ satisfait $\bigwedge_{p' \in X \setminus \mathcal{M}} \overline{p'}$, alors on obtient $\mathcal{M}' \cap X \subseteq \mathcal{M} \cap X$. Autrement, $\mathcal{M}'$ satisfait $\bigvee_{p \in \mathcal{M} \cap X} \overline{p}$ et cela signifie que $(\mathcal{M} \cap X) \setminus (\mathcal{M}' \cap X) \neq \emptyset$. Cette dernière propriété exprime que soit $\mathcal{M}' \cap X \subset \mathcal{M} \cap X$ soit $\mathcal{M}$ et $\mathcal{M}'$ sont incomparable selon $\preceq_X$.

Soient Φ une formule propositionnelle, X un ensemble de variables propositionnelles et $\mathcal{M}$ un modèle de Φ. Alors, $\mathcal{M}$ est un *modèle X-minimal* de Φ si il n'existe pas de modèle strictement plus petit que $\mathcal{M}$ selon $\preceq_X$. Dans [5], il a été démontré que trouver un modèle X-minimal est $P^{NP[O(log(n))]}$-difficile, où n est le nombre de variables propositionnelles.

L'ensemble des modèles X-minimaux correspond à l'ensemble de tous les modèles top-1 selon $\preceq_X$ et $Var(\Phi)$. En effet, si $\mathcal{M}$ est un modèle top-1, alors il n'existe pas de modèle $\mathcal{M}'$ tel que $\mathcal{M}' \prec_X \mathcal{M}$, et cela signifie que $\mathcal{M}$ est un modèle X-minimal. Dans ce contexte, notons que calculer l'ensemble des modèles top-k pour $k \geq 1$ peut être vu comme une généralisation du problème du calcul des modèles X-minimaux.

3.3 Un algorithme générique pour Top-k SAT

Dans cette section, nous décrivons notre algorithme pour le problème Top-k SAT dans le cas des relations de δ-préférences (Algorithme 1). L'idée de base est tout simplement d'utiliser la formule $f_{\succeq}(\mathcal{M})$ associée à un modèle $\mathcal{M}$ afin d'obtenir des modèles qui sont au moins aussi préférés que $\mathcal{M}$. Cet algorithme prend en entrée une formule CNF Φ, une relation de δ-préférences $\succeq$, un entier naturelle $k \geq 1$, et un ensemble X de variables propositionnelles permettant de

Algorithme 1: Top-k

Input : Une formule CNF Φ, une relation de préférences $\succeq$, un entier $k \geq 1$, et un ensemble X de variables
Output : Un ensemble $\mathcal{L}$ de modèle top-k

```
1  Φ' ← Φ;
2  L ← ∅
3  while (solve(Φ')) do            /* M est un modèle de Φ' */
4  |  if (∃M' ∈ L.M ≈_X M' & M ≻ M') then
5  |  |  replace(M, M', L);
6  |  else if (∀M' ∈ L.M ≉_X M' & |preferred(M, L)| < k)
   |  then
7  |  |  S ← min_top(k, L);
8  |  |  add(M, L);
9  |  |  remove(k, L);
10 |  |  S ← min_top(k, L) \ S;
11 |  |  Φ' ← Φ' ∧ ⋀_{M'∈S} f_≥(M');
12 |  else
13 |  |  Φ' ← Φ' ∧ f_≥(M)
14 |  Φ' ← Φ' ∧ ¬M;
15 return L;
```

définir la relation d'équivalence $\approx_X$. Il fournit en sortie un ensemble $\mathcal{L}$ de modèles top-k de Φ vérifiant les deux propriétés données dans la définition du problème Top-k SAT.

3.3.1 Description de l'algorithme

Dans la boucle *while*, nous utilisons des bornes inférieures pour trouver les modèles optimaux. Ces bornes inférieures sont obtenues en utilisant le fait que la relation de préférences considérée est une relation de δ-préférences. À chaque étape, une borne inférieures est intégrée en utilisant la formule :

$$\bigwedge_{\mathcal{M}' \in S} f_{\succeq}(\mathcal{M}')$$

- **Lingnes 4 – 5.** Notons premièrement que la procédure `replace(`$\mathcal{M}$`,`$\mathcal{M}'$`,`$\mathcal{L}$`)` remplace $\mathcal{M}'$ avec $\mathcal{M}$ dans $\mathcal{L}$. Nous appliquons ce remplacement car il existe un modèle $\mathcal{M}'$ dans $\mathcal{L}$ qui équivalent à $\mathcal{M}'$ et $\mathcal{M}$ permet d'obtenir une meilleure borne inférieure.
- **Lignes 6 – 11.** Dans le cas où $\mathcal{M}$ n'est équivalent à aucun modèle dans $\mathcal{L}$ et le nombre de modèles dans $\mathcal{L}$ qui lui sont préférés est strictement inférieur à k ($|\texttt{preferred}(\mathcal{M}, \mathcal{L})| < k$), nous ajoutons $\mathcal{M}$ à $\mathcal{L}$ (`add`($\mathcal{M}$,$\mathcal{L}$)). Notons que $\mathcal{S}$ contient premièrement les modèles de $\mathcal{L}$ avant l'ajout de $\mathcal{M}$ qui ont exactement $k-1$ modèles qui leur sont préférés dans cet ensemble. Avant l'ajout de $\mathcal{M}$ à $\mathcal{L}$, nous supprimons de $\mathcal{L}$ les modèles qui ne sont pas top-k, i.e., ils ont plus que $k-1$ modèles qui leur sont préférés dans $\mathcal{L}$ (`remove`$(k, \mathcal{L})$).
  Ensuite, nous modifions le contenu de S. Il est à noter que les éléments de S avant l'ajout de $\mathcal{M}$ ont été utilisés comme des bornes inférieures dans des étapes précédentes. Par conséquent, afin d'éviter l'ajout de la même borne plusieurs fois, le nouveau contenu de S correspond aux modèles dans $\mathcal{L}$ qui ont exactement $k-1$ modèles qui leurs sont préférés dans $\mathcal{L}$ (`min_top`$(k, \mathcal{L})$) privé des éléments du précédent contenu de S. Dans la ligne 11, nous intégrons des bornes inférieures dans Φ' en utilisant les éléments de S. Effectivement, pour tout modèle $\mathcal{M}$ de $\Phi' \wedge \bigwedge_{\mathcal{M}' \in S} f_{\succeq}(\mathcal{M}')$, on a $\mathcal{M}' \not\succ \mathcal{M}$, pour tout $\mathcal{M}' \in S$.
- **Lines 12 – 13.** Dans le cas où $\mathcal{M}$ n'est pas un modèle top-k, nous intégrons la borne inférieure qui lui est associée.
- **Line 14.** Cette instruction permet d'éviter de trouver le même modèle dans deux étapes différentes de la boucle *while*.

Proposition 1 *L'algorithme Top-k (Algorithme 1) est correct.*

Démonstration : La démonstration de la correction est fondée sur la définition des relations de δ-préférences. En effet, la fonction $f_{\succeq}$ nous permet d'exploiter des bornes afin d'améliorer systématiquement le niveau de préférences des modèles. Comme le nombre des modèles est fini, l'ajout des négations des modèles trouvés à chaque itération mène à une formule incohérente. Par conséquent, l'algorithme termine.

Comme expliquer dans la description, nous utilisons des bornes inférieures pour trouver les modèles optimaux. Ces bornes sont obtenues en utilisant la fonction $f_{\succeq}$.

□

4 Relations de préférences totales

Nous fournissons ici un second algorithme pour le calcul des modèles top-k dans le cas des relations de δ-préférences totales (Algorithme 2). Une relation de δ-préférences $\succeq$ est totale si, pour tous modèles $\mathcal{M}$ et $\mathcal{M}'$, on a $\mathcal{M} \succeq \mathcal{M}'$ ou $\mathcal{M}' \succeq \mathcal{M}$.

Notre algorithme dans ce cas est fourni dans Algorithme 2 :

- **Lignes 3 – 8**. Dans cette partie, nous calculons l'ensemble $\mathcal{L}$ de k modèles différents de Φ tel que, pour tout $\mathcal{M}, \mathcal{M}' \in \mathcal{L}$ avec $\mathcal{M} \neq \mathcal{M}'$, on a $\mathcal{M} \not\approx_X \mathcal{M}'$. En effet, si $\mathcal{M}$ est un modèle de Φ et $\mathcal{M}'$ est un modèle de $\Phi \wedge \overline{\mathcal{M}^1_{|X}} \wedge \cdots \wedge \overline{\mathcal{M}^n_{|X}} \wedge \overline{\mathcal{M}_{|X}}$, alors on peut aisément déduire que $\mathcal{M} \not\approx_X \mathcal{M}'$.
- **Ligne 9**. L'ensemble $min(\mathcal{L})$ correspond au plus grand sous-ensemble de $\mathcal{L}$ vérifiant la propriété suivante : pour tout $\mathcal{M} \in min(\mathcal{L})$, il n'existe pas de modèle dans $\mathcal{L}$ qui est strictement moins préféré que $\mathcal{M}$. L'affectation dans cette ligne nous

Algorithme 2: Top-k^T

Input : Une formule CNF Φ, une relation totale de préférences $\succeq$, un entier $k \geq 1$, et un ensemble X de variables
Output : Un ensemble $\mathcal{L}$ de modèle top-k

```
1  Φ' ← Φ;
2  L ← ∅
3  for (i ← 0 to k − 1) do
4  |  if (solve(Φ')) then
5  |  |  add(M, L);                    /* M est un modèle Φ' */
6  |  |  Φ' ← Φ' ∧ ¬M_|X;
7  |  else
8  |  |  return L;
9  Φ' ← Φ ∧ ⋀_{M∈L} ¬M ∧ ⋀_{M'∈min(L)} f_≻(M');
10 while (solve(Φ')) do              /* M est un modèle de Φ' */
11 |  if (∃M' ∈ L.M ≈_X M' & M ≻ M') then
12 |  |  replace(M, M', L);
13 |  else if (∀M' ∈ L.M ≉_X M') then
14 |  |  S ← min(L);
15 |  |  add(M, L);
16 |  |  remove(k, L);
17 |  |  S ← min(L) \ S;
18 |  |  Φ' ← Φ' ∧ ⋀_{M'∈S} f_≻(M');
19 |  |  else
20 |  |  |  Φ' ← Φ' ∧ f_≻(M)
21 |  Φ' ← Φ' ∧ ¬M;
22 return L;
```

permet d'obtenir uniquement des modèles qui sont au moins aussi préférés qu'un élément de $min(\mathcal{L})$. En effet, nous n'avons pas besoin de considérer les modèles qui sont moins préférés que les éléments de $min(\mathcal{L})$, car il est clair qu'ils ne sont pas des modèles top-k. Notons que tous les éléments de $min(\mathcal{L})$ sont équivalents par rapport à la relation $\approx$ induite par $\succeq$, car cette relation est totale.

— **Line 10 − 21**. La boucle *while* est similaire à celle de l'algorithme Top-k. Nous supprimons seulement la condition $|preferred(\mathcal{M}, \mathcal{L})| < k$ et nous remplaçons $min\_top(k, \mathcal{L})$ avec $min(\mathcal{L})$. En fait, étant donné que la relation $\succeq$ est totale, il est clair que l'on a $|preferred(\mathcal{M}, \mathcal{L})| < k$ par les bornes inférieures ajoutées précédemment. En outre, comme $\succeq$ est totale, l'ensemble des modèles supprimés par $remove(k, \mathcal{L})$ (Ligne 16) est soit un ensemble vide, soit $min(\mathcal{L})$.

Proposition 2 *L'algorithme Top-k^T (Algorithme 2) est correct.*

La correction de cet algorithme est obtenue à partir de celle de l'algorithme Top-k et le fait que les relations de δ-préférences considérées sont totales.

5 Une application de Top-k SAT en fouille de données

Le problème de l'extraction des motifs ensemblistes fréquents est fondamental en fouille de donnée [1]. De nombreuse tâche de fouille de données sont liées à ce problème comme celui de l'extraction de règles d'association. Récemment, les auteurs de [17, 11] ont proposé la première méthode fondée sur la programmation par contraintes (CP) pour le problème de l'extraction des motifs ensemblistes fréquents. Ce cadre fournit une approche déclarative et flexible. Il permet, entre autres, aux problèmes de fouille de données de bénéficier de plusieurs techniques de résolution dans la programmation par contraintes.

Dans le problème de l'extraction des motifs ensemblistes, la notion de motif top-k est introduite comme une alternative à l'utilisation de seuils de fréquence. Dans cette section, nous proposons un encodage permettant l'énumération des motifs ensemblistes fermés. Ensuite, nous utilisons cet encodage dans le cadre du problème Top-k SAT pour le calcul des motifs top-k.

5.1 Extraction des motifs top-k

Soit $\mathcal{I}$ un ensemble d'items. Une *transaction* correspond à un couple (tid, I) où tid est son identifiant et I est un ensemble d'items ($I \subseteq \mathcal{I}$). Une *base de transactions* est un ensemble fini de transactions où chaque identifiant ne réfère qu'une seule transaction. On dit qu'une transaction (tir, I) supporte un ensemble d'items J si $J \subseteq I$.

La *couverture* d'un ensemble d'items I dans une base de transaction $\mathcal{D}$ est l'ensemble des identifiants des transactions dans $\mathcal{D}$ supportant I : $\mathcal{C}(I, \mathcal{D}) = \{tid \mid (tid, J) \in \mathcal{D}, I \subseteq J\}$. Le *support* de I dans $\mathcal{D}$ est défini par : $\mathcal{S}(I, \mathcal{D}) = \mid \mathcal{C}(I, \mathcal{D}) \mid$. En outre, la *fréquence* de I dans $\mathcal{D}$ est : $\mathcal{F}(I, \mathcal{D}) = \frac{\mathcal{S}(I,\mathcal{D})}{|\mathcal{D}|}$.

Par exemple, considérons la base de transactions suivante :

tid	ensemble d'items
1	a, b, c, d
2	a, b, e, f
3	a, b, c, m
4	a, c, d, f, j
5	j, l
6	d
7	d, j

Transaction database $\mathcal{D}$

Dans cette base, on a $\mathcal{S}(\{a, b, c\}, \mathcal{D}) = |\{1, 3\}| = 2$ et $\mathcal{F}(\{a, b\}, \mathcal{D}) = \frac{3}{7}$.

Soient $\mathcal{D}$ une base de transactions définie sur $\mathcal{I}$ et λ un seuil de support. Le problème de l'extraction des ensembles fréquents consiste à calculer l'ensemble suivant :

$$\mathcal{FIM}(\mathcal{D}, \lambda) = \{I \subseteq \mathcal{I} \mid \mathcal{S}(I, \mathcal{D}) \geq \lambda\}$$

Définition 3 (Motif fermé) *Soient $\mathcal{D}$ une base de transactions et I un ensemble d'items tel que $\mathcal{S}(I,\mathcal{D}) \geq 1$. I est fermé ssi, pour tout ensemble d'items J tel que $I \subset J$, $\mathcal{S}(J,\mathcal{D}) < \mathcal{S}(I,\mathcal{D})$.*

On peut aisément constater que tous les motifs fréquents peuvent être obtenus à partir des motifs fermés en calculant leurs sous-ensembles. Étant donné que le nombre des motifs fermés est inférieur à celui des motifs fréquents, l'énumération des motifs fermés permet de réduire la taille de la sortie.

Dans cet article, nous considérons principalement le problème de l'énumération des motifs ensemblistes fermés top-k ayant des tailles supérieures à min. Dans ce problème, nous considérons le seuil de fréquence comme étant inconnu.

Définition 4 ($\mathcal{FCIM}^k_{min}$) *Soient k et min deux nombres naturels strictement supérieurs à 0. Le problème de l'énumération des motifs fermés top-k consiste à calculer tous les motifs fermés ayant des tailles supérieures ou égales à min tels que, pour chaque motif, il n'existe pas plus de $k-1$ motifs fermés ayant des tailles supérieures ou égales à min et des valeurs de support supérieures à la sienne.*

5.2 Un encodage fondé sur SAT pour $\mathcal{FCIM}^k_{min}$

Nous proposons ici un encodage fondé sur SAT pour le problème $\mathcal{FCIM}^k_{min}$. Soient $\mathcal{I}$ un ensemble d'items, $\mathcal{D} = \{(0,t_i),\ldots,(n-1,t_{n-1})\}$ une base de transactions définie sur $\mathcal{I}$, et k et min deux nombre naturels strictement supérieurs à 0. On associe à chaque item dans $\mathcal{D}$ une variable propositionnelle p_a. De telles variables propositionnelles permettent d'encoder l'ensemble candidat $I \subseteq \mathcal{I}$, i.e., $p_a = 1$ ssi $a \in I$. De plus, pour chaque $i \in \{0,\ldots,n-1\}$, on associe à la i-ème transaction une variable propositionnelle b_i.

Dans un premier temps, nous introduisons la contrainte permettant de considérer uniquement les ensembles ayant des tailles supérieures ou égales à min :

$$\sum_{a\in\mathcal{I}} p_a \geq min \tag{1}$$

Nous introduisons maintenant la contrainte permettant de capturer les transactions où le candidat n'est pas supporté :

$$\bigwedge_{i=0}^{n-1} (b_i \leftrightarrow \bigvee_{a\in\mathcal{I}\setminus t_i} p_a) \tag{2}$$

Cette contrainte signifie que b_i est vrai si et seulement si le candidat n'est pas supporté par t_i.

Par la contrainte suivante, nous forçons le candidat à être fermé :

$$\bigwedge_{a\in\mathcal{I}} (\bigwedge_{i=0}^{n-1} \overline{b_i} \rightarrow a \in t_i) \rightarrow p_a \tag{3}$$

Intuitivement, cette contrainte signifie que si $\mathcal{S}(I,\mathcal{D}) = \mathcal{S}(I\cup\{a\},\mathcal{D})$ alors on a $a \in I$.

Dans ce contexte, le calcul des motifs ensemblistes fermés top-k de tailles supérieures à min correspond au calcul des modèles top-k de (1), (2) et (3) selon la relation $\succeq_B$ et $X = \{p_a | a \in \mathcal{I}\}$, où $B = \{b_0,\ldots,b_{n-1}\}$ et $\succeq_B$ est définie comme suit : $\mathcal{M} \succeq_B \mathcal{M}'$ si et seulement si $|\mathcal{M}(B)| \leq |\mathcal{M}'(B)|$. Cette relation est une relation de δ-préférences, car on peut définir $f_{\succeq_B}$ comme suit :

$$f_{\succeq_B}(\mathcal{M}) = (\sum_{i=0}^{n-1} b_i \leq |\mathcal{M}(B)|)$$

En effet, cette formule nous permet d'avoir des modèles correspondant à des motifs fermés avec des supports supérieurs ou égaux à celui de l'ensemble associé à $\mathcal{M}$.

5.3 Quelques variantes de $\mathcal{FCIM}^k_{min}$

Notre but ici est d'illustrer les bons aspects déclaratifs de notre cadre. Ainsi, nous considérons des variantes simples de $\mathcal{FCIM}^k_{min}$ et nous montrons que leurs encodages fondés sur SAT peuvent être obtenus à l'aide de modifications simples :

5.3.1 Variante 1 ($\mathcal{FCIM}^k_{max}$)

Dans cette variante, nous considérons le problème de l'énumération des motifs fermés top-k de tailles inférieures à max. Un encodage pour cette variante peut être obtenu en ajoutant à (2) et (3) la contrainte suivante :

$$\sum_{a\in\mathcal{I}} p_a \leq max \tag{4}$$

Dans ce cas, nous utilisons la même relation de δ-préférences $\succeq_B$ définie précédemment.

5.3.2 Variante 2 ($\mathcal{FCIM}^k_{\lambda}$)

Nous proposons maintenant un encodage du problème de l'énumération des motifs fermés top-k ayant des supports supérieurs à λ. Dans ce contexte, un motif fermé top-k est un motif fermé ayant un support supérieur à λ tel qu'il existe moins de $k-1$ motifs fermés avec des supports supérieurs à λ et de tailles strictement supérieures à la sienne. Notre encodage, dans ce cas, est obtenu par l'ajout à (2) et (3) de la contrainte suivante :

$$\sum_{i=0}^{n} \overline{b_i} \geq \lambda \quad (5)$$

La relation de préférences utilisée est $\succeq_I$ définie comme suit : $\mathcal{M} \succeq_I \mathcal{M}'$ si et seulement si $|\mathcal{M}(I)| \geq |\mathcal{M}'(I)|$. Elle est une relation de δ-préférences car $f_{\succeq_I}$ peut être définie de la manière suivante :

$$f_{\succeq_I}(\mathcal{M}) = \sum_{a \in I} p_a \geq |\mathcal{M}(I)|$$

5.3.3 Variante 3 ($\mathcal{FMIM}_{\lambda}^{k}$)

Nous considérons ici une variante du problème de l'énumération des motifs maximaux. Elle consiste à calculer les motifs maximaux top-k ayant des supports supérieurs à λ et pour chacun d'eux, il n'existe pas plus de $k-1$ motifs maximaux avec des tailles supérieures. Notre encodage correspond aux deux contraintes (2) et (5). Nous utilisons dans ce cas la relation de δ-préférences $\succeq_I$.

6 Expérimentations

Dans cette section, nous évaluons les performances de notre algorithme pour Top-k SAT de manière empirique. Notre premier objectif est de démontrer les bons aspects déclaratifs et l'efficacité de notre cadre. Nous considérons dans ce contexte le problème $\mathcal{FCIM}_{min}^{k}$.

Pour nos expérimentations, nous avons implanté l'algorithme Top-k (Algorithme 1) en utilisant le logiciel MiniSAT 2.2 [1]. Dans notre encodage fondé sur SAT, nous avons utilisé l'encodage de la contrainte de cardinalité proposé dans [8].

Nous avons considéré une variété d'instances obtenue à partir des dépôts `FIMI` [2] et `CP4IM` [3]. Toutes les expérimentations ont été effectuées sur des machines Intel Xeon quad-core avec 32GB de RAM cadencé à 2,66 Ghz. Pour chaque instance, nous avons utilisé un temps mort de 4 heures de temps CPU.

Table 1 fournit des détails sur les différentes bases de transactions. La première colonne contient le nom de l'instance considérée. Dans la deuxième et la troisième colonnes nous fournissons la taille de l'instance en termes de respectivement le nombre de transactions (#trans) et le nombre d'items (#items). La quatrième colonne contient la densité de la base (dans) définie comme le pourcentage de 1 dans la base. L'ensemble des instances varie des peu danses (e.g. mushroom) au très denses (e.g. Hepatitis). Enfin, dans les deux dernières colonnes, nous fournissons la taille de la formule CNF (#vars, #clauses) qui correspond à notre encodage de $\mathcal{FCIM}_{min}^{k}$. On peut dores et déjà constater que ces formules ont des tailles raisonnables. La taille maximale est obtenue dans le cas de l'instance *connect* (67 815 variables et 5 877 720 clauses).

instance	#trans	#items	$dens(\%)$	#vars	#clauses
zoo-1	101	36	44	173	2196
Hepatitis	137	68	50	273	4934
Lymph	148	68	40	284	6355
audiology	216	148	45	508	17575
Heart-cleveland	296	95	47	486	15289
Primary-tumor	336	31	48	398	5777
Vote	435	48	33	531	14454
Soybean	650	50	32	730	22153
Australian-credit	653	125	41	901	48573
Anneal	812	93	45	990	39157
Tic-tac-toe	958	27	33	1012	18259
german-credit	1000	112	34	1220	73223
Kr-vs-kp	3196	73	49	3342	121597
Hypothyroid	3247	88	49	3419	143043
chess	3196	75	49	3346	124797
splice-1	3190	287	21	3764	727897
mushroom	8124	119	18	8348	747635
connect	67558	129	33	67815	5877720

TABLE 1 – Caractéristiques des instances

Afin d'analyser le comportement de notre algorithme pour Top-k SAT sur $\mathcal{FCIM}_{min}^{k}$, nous avons réalisé deux types d'expérimentations. Dans le premier, nous donnons à *min* la valeur 1 alors que l'on varie la valeur de k de 1 à 10000. Dans le second type, nous fixons la valeur de k à 10 et on varie la valeur de *min* de 1 jusqu'à la taille maximale.

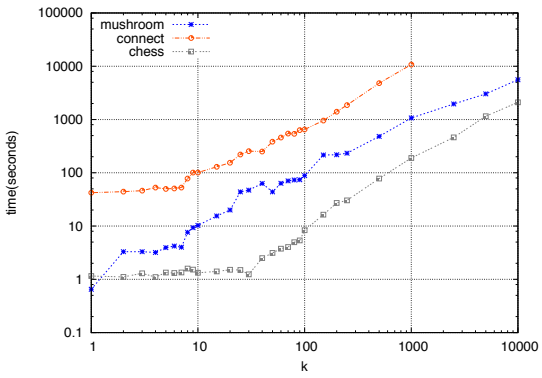


FIGURE 1 – $\mathcal{FCIM}_{1}^{k}$ pour des valeurs différentes de k

Des résultats pour un ensemble d'instances représentatives sont fournis en Figure 1 (échelle logarithmique). Les autres instances présentent un comportement similaire. Comme attendu, le temps nécessaire pour le calcul des modèles top-k augmente avec l'augmentation de la valeur k. Pour *connect*, notre algorithme ne parvient pas dans la limite du temps mort à calculer tout les modèles top-k pour les valeurs de k supérieures à 1000. Cette figure montre clairement que le

1. MiniSAT : http ://minisat.se/
2. FIMI : http ://fimi.ua.ac.be/data/
3. CP4IM : http ://dtai.cs.kuleuven.be/CP4IM/datasets/

calcul des modèles top-k peut être réalisé de manière efficace pour des valeurs petites de k. Par exemple, sur toutes les instances, le calcul des modèles top-10 nécessite moins de 100 seconds. En conséquence, il est évident que le problème de l'extraction des motifs top-k offre une alternative à l'utilisateur dans le contrôl de la taille de la sortie. Dans Figure 2, nous montrons que les résultats obtenus de l'instance la plus dure de Table 1. Sur `splice-1`, l'algorithme ne parvient pas à tout calculer dans la limite du temps mort pour $k > 20$.

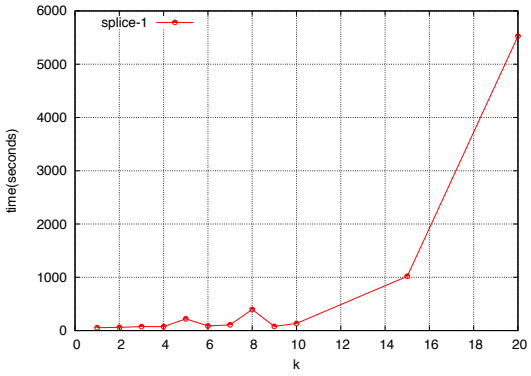


FIGURE 2 – Instance de $\mathcal{FCIM}_1^k$ difficile

Dans notre second type d'expérimentations, notre objectif est d'étudier le comportement de notre algorithme lorsque l'on varie la valeur de min. Dans Figure 3, nous fournissons des résultats obtenus sur trois instances représentatives (mushroom, connect et chess) où k est fixé à 10 et min varie entre 1 jusqu'à la taille maximale des transactions. Notre algorithme est efficace dans le cas où problème est sous-contraint (valeurs petites pour min - de nombreux modèles top-k) mais également dans le cas où le problème est sur-contraint (valeurs grandes pour min - un petit nombre de modèles top-k). Quant à l'instance `connect`, l'algorithme ne parvient pas à résoudre le problème dans la limite du temps mort pour $min > 15$. Pour toutes les instances, les différentes courbes présentent un pic de difficulté pour les valeurs moyennes de min.

7 Conclusion and Perspectives

Dans cet article, nous introduisons un nouveau problème, appelé Top-k SAT, défini comme le problème d'énumération des modèles top-k d'une formule propositionnelle. Un modèle top-k est défini comme un modèle ayant moins de k modèles qui lui sont préférés par rapport à une relation de préférence. Nous montrons que le problème Top-k SAT généralise deux autres problèmes connus : le problème Partial Max-SAT et le

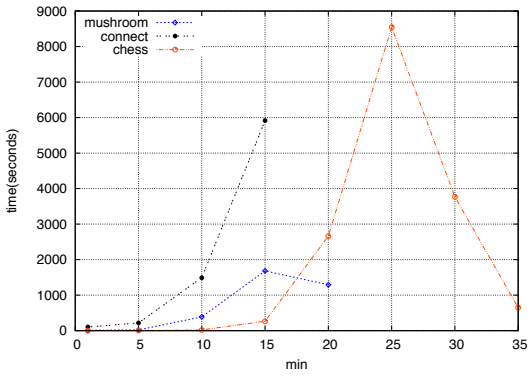


FIGURE 3 – $\mathcal{FCIM}_{min}^{10}$ pour des valeurs différentes de min

problème des modèles minimaux. Un algorithme générique pour Top-k SAT est proposé et évalué de manière expérimentale sur le problème de l'extraction des motifs fermé top-k.

Bien que notre nouveau problème de calcul des modèles préférés top-k est flexible et déclarative, il y a un certain nombre de questions qui méritent des efforts de recherche supplémentaires. Une des directions est l'amélioration en terme d'efficacité de l'algorithme permettant l'énumération des modèles top-k. On envisage également l'étude de l'application du problème Top-k SAT dans le cadre d'autres problème de fouille de données, comme la fouille de séquences.

Références

[1] R. Agrawal, T. Imielinski, and A. N. Swami. Mining association rules between sets of items in large databases. In *ACM SIGMOD International Conference on Management of Data*, pages 207–216, Baltimore, 1993. ACM Press.

[2] Olivier Bailleux and Yacine Boufkhad. Efficient CNF Encoding of Boolean Cardinality Constraints. In *9th International Conference on Principles and Practice of Constraint Programming - CP 2003*, pages 108–122, 2003.

[3] Craig Boutilier, Ronen I. Brafman, Carmel Domshlak, David L. Poole, and Holger H. Hoos. CP-nets : A Tool for Representing and Reasoning with Conditional Ceteris Paribus Preference Statements. *Journal of Artificial Intelligence Research (JAIR)*, 21 :135–191, 2004.

[4] R. I. Brafman and Carmel Domshlak. Preference Handling - An Introductory Tutorial. *AI Magazine*, 30(1) :58–86, 2009.

[5] Marco Cadoli. On the complexity of model finding for nonmonotonic propositional logics. In *4th Italian conference on theoretical computer science*, pages 125–139, 1992.

[6] Thierry Castell, Claudette Cayrol, Michel Cayrol, and Daniel Le Berre. Using the davis and putnam procedure for an efficient computation of preferred models. In *ECAI*, pages 350–354, 1996.

[7] Carmel Domshlak, Eyke Hüllermeier, Souhila Kaci, and Henri Prade. Preferences in AI : An overview. *Artificial Intelligence*, 175(7-8) :1037–1052, 2011.

[8] N. Eén and N. Sörensson. Translating pseudo-boolean constraints into SAT. *JSAT*, 2(1-4) :1–26, 2006.

[9] Ada Wai-Chee Fu, Renfrew W. w. Kwong, and Jian Tang. Mining n-most interesting itemsets. In *Proceedings of the 12th International Symposium on Methodologies for Intelligent Systems (ISMIS 2000)*, Lecture Notes in Computer Science, pages 59–67. Springer, 2000.

[10] Zhaohui Fu and Sharad Malik. On Solving the Partial MAX-SAT Problem. In *Proceedings of the Ninth International Conference on Theory and Applications of Satisfiability Testing (SAT'06)*, pages 252–265, 2006.

[11] Tias Guns, Siegfried Nijssen, and Luc De Raedt. Itemset mining : A constraint programming perspective. *Artificial Intelligence*, 175(12-13) :1951–1983, August 2011.

[12] Jiawei Han, Jianyong Wang, Ying Lu, and Petre Tzvetkov. Mining top-k frequent closed patterns without minimum support. In *Proceedings of the 2002 IEEE International Conference on Data Mining (ICDM 2002)*, pages 211–218. IEEE Computer Society, 2002.

[13] Yiping Ke, James Cheng, and Jeffrey Xu Yu. Top-k correlative graph mining. In *Proceedings of the SIAM International Conference on Data Mining (SDM 2009)*, pages 1038–1049, 2009.

[14] Hoang Thanh Lam and Toon Calders. Mining top-k frequent items in a data stream with flexible sliding windows. In *Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD 2010)*, pages 283–292, 2010.

[15] Hoang Thanh Lam, Toon Calders, and Ninh Pham. Online discovery of top-k similar motifs in time series data. In *Proceedings of the Eleventh SIAM International Conference on Data Mining, SDM 2011 (SDM 2011)*, pages 1004–1015, 2011.

[16] P. Meseguer, F. Rossi, and T. Schiex. *Soft Constraints*, chapter 9. Elsevier, 2006.

[17] Luc De Raedt, Tias Guns, and Siegfried Nijssen. Constraint programming for itemset mining. In *ACM SIGKDD*, pages 204–212, 2008.

[18] Emanuele Di Rosa, Enrico Giunchiglia, and Marco Maratea. Solving satisfiability problems with preferences. *Constraints*, 15(4) :485–515, 2010.

[19] Yoav Shoham. *Reasoning about change : time and causation from the standpoint of artificial intelligence*. MIT Press, Cambridge, MA, USA, 1988.

[20] Carsten Sinz. Towards an optimal cnf encoding of boolean cardinality constraints. In *11th International Conference on Principles and Practice of Constraint Programming - CP 2005*, pages 827–831, 2005.

[21] A. Tiwari, R.K. Gupta, and D.P. Agrawal. A survey on frequent pattern mining : Current status and challenging issues. *Inform. Technol. J*, 9 :1278–1293, 2010.

[22] G.S. Tseitin. On the complexity of derivations in the propositional calculus. In *Structures in Constructives Mathematics and Mathematical Logic, Part II*, pages 115–125, 1968.

[23] Elena Valari, Maria Kontaki, and Apostolos N. Papadopoulos. Discovery of top-k dense subgraphs in dynamic graph collections. In *Proceedings of the 24th International Conference on Scientific and Statistical Database Management (SSDBM 2012)*, pages 213–230, 2012.

[24] Toby Walsh. Representing and reasoning with preferences. *AI Magazine*, 28(4) :59–70, 2007.

[25] Jianyong Wang, Jiawei Han, Ying Lu, and Petre Tzvetkov. TFP : An efficient algorithm for mining top-k frequent closed itemsets. *IEEE Transactions on Knowledge Data Engineering*, 17(5) :652–664, 2005.

[26] Joost P. Warners. A linear-time transformation of linear inequalities into conjunctive normal form. *Information Processing Letters*, 1996.

ACID : Disjonction constructive adaptative sur intervalles

Bertrand Neveu[1] **Gilles Trombettoni**[2]

[1] LIGM, Université Paris Est, France
[2] LIRMM, Université Montpellier 2, France
Bertrand.Neveu@enpc.fr Gilles.Trombettoni@lirmm.fr

Résumé

Un opérateur de filtrage nommé CID et une variante efficace 3BCID ont été proposés en 2007. Ces opérateurs calculent pour des CSP numériques traités avec des méthodes à intervalles une consistance partielle équivalente à Partition-1-AC pour les CSP à domaines finis. Les deux paramètres principaux de CID sont le nombre d'appels à la procédure principale et le nombre maximum de sous-intervalles traités par cette procédure. Cet opérateur 3BCID est efficace pour la résolution de CSP numériques, mais l'est moins en optimisation globale sous contraintes.

Cet article propose une variante adaptative de 3BCID. Le nombre de variables traitées est calculé automatiquement durant la recherche, les autres paramètres étant fixés de manière robuste. Sur un échantillon représentatif d'instances, ACID apparaît comme la meilleure approche en résolution et en optimisation. Elle a été choisie comme méthode de contraction par défaut du résolveur par intervalles Ibex.

Abstract

An operator called CID and an efficient variant 3BCID were proposed in 2007. For numerical CSPs handled by interval methods, these operators compute a partial consistency equivalent to Partition-1-AC for discrete CSPs. The main two parameters of CID are the number of times the main CID procedure is called and the maximum number of sub-intervals treated by the procedure. The 3BCID operator is state-of-the-art in numerical CSP solving, but not in constrained global optimization.

This paper proposes an adaptive variant of 3BCID. The number of variables handled is auto-adapted during the search, the other parameters are fixed and robust to modifications. On a representative sample of instances, ACID appears to be the best approach in solving and optimization, and has been added to the default strategies of the Ibex interval solver.

1 Disjonction constructive sur intervalles (CID)

Un opérateur de filtrage/contraction pour les *CSP numériques* appelé *Constructive Interval Disjunction* (CID) a été proposé dans [13]. Appliqué d'abord aux problèmes de satisfaction traités par les méthodes à intervalles, l'opérateur a été appliqué plus récemment en optimisation globale sous contraintes. Cet algorithme détient les performances de référence en résolution, mais est généralement surpassé en optimisation par les algorithmes plus simples de propagation de contraintes comme HC4. La principale contribution pratique de cet article est de montrer qu'une version auto-adaptative de CID peut devenir efficace à la fois en résolution et en optimisation, et ce, sans ajouter de paramètres utilisateurs.

1.1 Rognage

Le principe de rognage (*shaving* en anglais) est utilisé pour calculer *Singleton Arc Consistency* (SAC) en domaines finis [7] et la 3B-consistance sur les CSP continus [9]. Le rognage est également au cœur de l'algorithme SATZ [11] utilisé pour prouver la satisfiabilité d'une formule booléenne. Le rognage fonctionne de la manière suivante. Une valeur est temporairement affectée à une variable (les autres valeurs étant temporairement supprimées du domaine) et une consistance est calculée sur le sous-problème correspondant. Si une inconsistance est détectée, alors la valeur peut être supprimée définitivement du domaine de la variable. Sinon la valeur est maintenue dans le domaine.

Contrairement à l'arc-consistance, cette consistance n'est pas incrémentale [7]. En effet, le travail de la procédure de réfutation porte sur toutes les variables d'un

sous-problème afin de supprimer une seule valeur d'un domaine. C'est pourquoi calculer SAC en domaines finis fait appel à un algorithme de point-fixe où toutes les variables doivent être traitées à nouveau à chaque retrait d'une valeur [7]. La remarque tient aussi pour la version améliorée `SAC-Opt` [5].

Le même principe de rognage peut être suivi pour les CSP numériques (NCSP).

1.2 CSP numérique

Un NCSP est défini par un triplet $P = (X, [X], C)$, où X désigne un n-ensemble de variables numériques, à valeurs réelles dans un domaine $[X]$. On note $[x_i] = [\underline{x_i}, \overline{x_i}]$ le domaine/intervalle de la variable $x_i \in X$, où $\underline{x_i}$ et $\overline{x_i}$ sont des nombres flottants (permettant d'implanter les algorithmes sur un ordinateur). Une solution de P est un n-vecteur dans $[X]$ qui satisfait toutes les contraintes de C. Les contraintes C définies dans un NCSP sont numériques. Ce sont des équations et des inégalités comprenant des opérateurs mathématiques comme $+$, $\cdot$, $/$, exp, log, sin.

On appelle *boîte* (parallèle aux axes) un produit cartésien d'intervalles, comme le domaine $[X] = [x_1] \times ... \times [x_n]$. $w(x_i)$ dénote la *largeur* $\overline{x_i} - \underline{x_i}$ d'un intervalle $[x_i]$. La largeur d'une boîte est donnée par la largeur $\overline{x_m} - \underline{x_m}$ de sa plus grande dimension x_m. L'union de plusieurs boîtes n'est généralement pas une boîte si bien que l'on utilise plutôt un opérateur d'enveloppe (*Hull*) pour calculer la plus petite boîte comprenant toutes les boîtes traitées.
Les NCSPs sont généralement résolus par une stratégie de type Brancher & Contracter à intervalles :

- **Brancher :** une variable x_i est choisie et son intervalle $[x_i]$ est divisé en deux sous-intervalles ; les deux sous-boîtes ainsi construites sont traitées (contractées), ce qui rend combinatoire l'ensemble du processus de résolution.
- **Contracter :** un processus de filtrage permet de contracter les intervalles (i.e., améliorer les bornes des intervalles) sans perte de solutions.

Le processus commence avec un domaine initial $[X]$ et s'arrête quand la largeur des feuilles/boîtes de l'arbre de recherche atteint une largeur inférieure à une précision donnée en entrée. Ces feuilles fournissent une approximation de toutes les solutions du NCSP.

Plusieurs algorithmes de contraction ont été proposés. Mentionnons l'algorithme de propagation de contraintes appelé `HC4` [3, 10], une implantation efficace de l'algorithme 2B [9] qui calcule une consistance locale optimale (enveloppe-consistance – *hull-consistency*) seulement si des hypothèses fortes sont réunies (en particulier, chaque variable doit apparaître au plus une fois dans une même contrainte). La procédure 2B-Revise travaille avec toutes les *fonctions de projection* d'une contrainte donnée. Informellement, une fonction de projection isole une occurrence donnée d'une variable dans l'expression de la contrainte. Par exemple, considérons la contrainte $x + y = z.x$; $x \leftarrow z.x - y$ est une fonction de projection (parmi d'autres) qui vise à réduire le domaine de la variable x. Evaluer la fonction de projection, en utilisant l'arithmétique des intervalles, sur le domaine $[x] \times [y] \times [z]$ (i.e., remplacer les occurrences de variable de la fonction de projection par leurs domaines et utiliser l'extension aux intervalles des opérateurs mathématiques impliqués) produit un intervalle image intersecté ensuite avec $[x]$. D'où une réduction potentielle du domaine. Une boucle de propagation proche de AC3 permet alors de propager les réductions obtenues pour un domaine de variable donné vers d'autres contraintes du système.

1.3 L'algorithme 3B

Des consistances plus fortes ont également été proposées. La 3B-consistance [9] est une consistance partielle similaire à SAC pour CSP quoique limitée aux bornes du domaine. Considérons les $2n$ sous-problèmes d'un NCSP donné où chaque intervalle $[x_i]$ ($i \in \{1..n\}$) est réduit à sa borne inférieure $\underline{x_i}$ (resp. borne supérieure $\overline{x_i}$). La 3B-consistance est vérifiée ssi chacun des $2n$ sous-problèmes est enveloppe-consistant.

En pratique, l'algorithme `3B`(w) subdivise les domaines en plusieurs sous-intervalles, également appelés tranches, de largeur w, largeur qui correspond à une précision : la 3B(w)-consistance est vérifiée ssi les tranches aux bornes de la boîte traitée ne peuvent pas être éliminées par `HC4`. Appelons `var3B` la procédure de l'algorithme de `3B` chargée de rogner un intervalle $[x_i]$. Le paramètre s_{3b} de `var3B` est un entier positif spécifiant un nombre de sous-intervalles : $w = w(x_i)/s_{3b}$ donne la largeur d'un sous-intervalle.

1.4 L'algorithme CID

La disjonction constructive sur intervalles (*Constructive Interval Disjunction* – `CID`) est une consistance plus forte que la 3B-consistance [13]. La CID-consistance est similaire à Partition-1-AC dans les CSP à domaines finis [4]. Partition-1-AC est strictement plus forte que SAC [4].

La procédure principale `varCID` traite une variable x_i. Les paramètres principaux de `varCID` sont x_i, un nombre s_{cid} de sous-intervalles (précision) et un algorithme de contraction *ctc*, comme `HC4`. $[x_i]$ est subdivisé en s_{cid} tranches de taille égale ; chaque sous-problème correspondant est traité par le contracteur *ctc* et on renvoie finalement l'enveloppe des différentes boîtes contractées, comme le détaille l'algorithme 1.

Procedure `VarCID` *(x_i, s_{cid}, (X, C, in-out $[X]$), ctc)*
- $[X]' \leftarrow$ *empty box*
- **for** $j \leftarrow 1$ *to* s_{cid} **do**
  - /* La j^e sous-boîte de $[X]$ sur x_i est traitée : */
  - `sliceBox` $\leftarrow$ `SubBox` (j, x_i, $[X]$)
  - /* Calcul d'une consistance sur la sous-boîte : */
  - `sliceBox'` $\leftarrow$ *ctc*(X, C, `sliceBox`)
  - /* "Union" avec les sous-boîtes précédentes : */
  - $[X]' \leftarrow$ `Hull`($[X]'$, `sliceBox'`)
- $[X] \leftarrow [X]'$

Algorithm 1: La procédure principale de l'opérateur `CID` pour le rognage du domaine d'une variable x_i.

Intuitivement, `CID` généralise `3B` puisqu'une sous-boîte éliminée par `var3B` serait aussi éliminée par `varCID`. De plus, contrairement à `var3B`, `varCID` peut aussi contracter $[X]$ sur *plusieurs* dimensions.

Notons que, dans l'implémentation effective, la boucle peut être interrompue plus tôt si $[X]'$ devient égale à la boîte initiale $[X]$ dans toutes les dimensions, exceptée x_i.

`var3BCID` est une variante (hybride) opérationelle de `varCID`.

1. Comme `var3B`, elle cherche d'abord à éliminer des sous-intervalles aux bornes de $[x_i]$, sous-intervalles de largeur $w = w(x_i)/s_{3b}$ chacun. On conserve les boîtes gauche $[X_l]$ et droite $[X_r]$ qui ne sont pas exclues par le contracteur *ctc* (s'il y en a).
2. Ensuite, la boîte restante $[X]'$ est traitée par `varCID` qui subdivise $[X]'$ en s_{cid} sous-boîtes. Les sous-boîtes sont contractés par *ctc*, leur enveloppe (hull) donnant $[X_{cid}]$.
3. Finalement, on renvoie l'enveloppe de $[X_l]$, $[X_r]$ et $[X_{cid}]$.

Le fonctionnement de `var3BCID` est illustré par la figure 1.

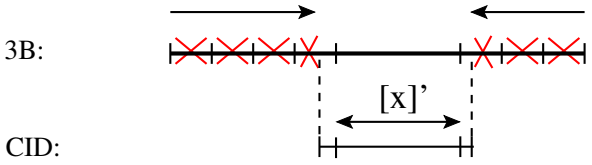


FIGURE 1 – Fonctionnement de la procédure `var3BCID`. La valeur 10 est choisie pour le paramètre s_{3b} et la valeur 1 est choisie pour s_{cid}.

`var3BCID` repose sur la volonté de gérer différentes largeurs de domaines (précisions) pour s_{3b} et s_{cid}. En effet, le meilleur choix pour s_{3b} se situe généralement dans la fourchette $\{5..20\}$ tandis que s_{cid} doit être quasiment toujours limité à 1 ou 2 (ce qui entraîne une enveloppe finale de 3 ou 4 sous-boîtes). La raison en est que le coût effectif en temps de la partie rognage est plus faible que celui de la disjonction constructive de domaine. En effet, si aucun sous-intervalle d'un domaine n'est éliminé par `var3B`, alors seulement deux appels à *ctc* sont réalisés, un pour chaque borne de l'intervalle traité ; alors que lorsque `varCID` est appliqué, le sous-contracteur est souvent appliqué s_{cid} fois.

La procédure `var3BCID` a été étudiée et expérimentée de manière approfondie dans le passé. Le nombre et l'ordre dans lequel les appels à `var3BCID` sont réalisés est une question plus difficile étudiée dans cet article.

2 CID adaptatif : apprendre le nombre de variables traitées

Comme dans `SAC` ou `3B`, un point quasi-fixe en terme de contraction peut être atteint par `3BCID` (ou `CID`) en appelant `var3BCID` dans deux boucles imbriquées. Une boucle intérieure appelle `var3BCID` sur chaque variable x_i. Une boucle extérieure appelle la boucle intérieure jusqu'à ce qu'aucun intervalle ne soit contracté plus que d'une certaine précision/largeur (atteignant ainsi un point quasi-fixe). Appelons `3BCID-fp` (*fixed-point*) cette version "historique".

Deux raisons nous ont amenés à changer radicalement cette gestion. D'abord, comme dit plus haut, `var3BCID` peut contracter la boîte traitée dans plusieurs dimensions. Un avantage significatif est que le point quasi-fixe en terme de contraction peut ainsi être atteint en un petit nombre d'appels à `var3BCID`. Sur la plupart des instances en satisfaction ou optimisation, il s'avère que le point quasi-fixe est obtenu en moins de n appels. Dans ce cas, `3BCID` est clairement trop coûteux. Ensuite, le principe derrière `varCID` est proche d'un point de choix dans un arbre de recherche. La différence est que l'on calcule une enveloppe des sous-boîtes obtenues après contraction (par *ctc*). C'est pourquoi une idée est d'utiliser une heuristique standard de branchement pour sélectionner la prochaine variable à "varcider". (Nous écrirons par la suite qu'une variable est *varcidée* quand la procédure `var3BCID` est appelée sur cette variable pour contracter la boîte courante.)

Pour résumer, l'idée pour rendre `3BCID` bien plus efficace en pratique est de remplacer les deux boucles imbriquées par une seule boucle appelant *numVarCID* fois la procédure `var3BCID` et d'utiliser une variante efficace de l'heuristique de branchement basée sur la fonction *Smear* pour sélectionner les variables à varcider (heuristique appelée `SmearSumRel` dans [12]). Informellement, la fonction Smear favorise

les variables avec un large domaine et un fort impact sur les contraintes – en mesurant les dérivées partielles (sur intervalles).

Une première idée est de fixer *numVarCID* au nombre n de variables. Appelons `3BCID-n` cette version. Elle produit de bons résultats en satisfaction mais est dominée par la propagation de contraintes pure en optimisation. Comme dit ci-dessus, cette approche est trop coûteuse quand la valeur optimale de *numVarCID* est inférieure à n (ce qui est souvent le cas en optimisation), mais peut aussi avoir un impact négatif sur les performances si un effort plus grand pouvait apporter un filtrage significativement plus grand.

Le but d'un algorithme `CID` adaptatif (`ACID`) est précisément de calculer dynamiquement pendant la recherche la valeur du paramètre *numVarCID*. Plusieurs politiques d'auto-adaptation ont été testées et nous décrivons trois versions intéressantes. Toutes ces politiques mesurent le gain en taille de l'espace de recherche après chaque appel à `var3BCID`. Ils mesurent un *ratio de contraction* d'une boîte $[X]^b$ par rapport à une boîte $[X]^a$ comme un gain moyen relatif dans toutes les dimensions :

$$\texttt{gainRatio}([X]^b,[X]^a) = \frac{1}{n}\sum_{i=1}^{n}(1-\frac{w(x_i^b)}{w(x_i^a)})$$

2.1 ACID0 : auto-adapter numVarCID pendant la recherche

La première version `ACID0` adapte le nombre de variables rognées dynamiquement à chaque nœud de l'arbre de recherche. Les variables sont tout d'abord triées selon leur impact, calculé par la même formule que pour l'heuristique de branchement `SmearSumRel`. Les variables sont ensuite rognées jusqu'à ce que le ratio de contraction cumulée pendant les nv derniers appels à `var3BCID` devienne inférieur à *ctratio*. Cet algorithme a donc 2 paramètres nv and *ctratio* qui se sont avérés assez difficiles à régler. Les expérimentations ont montré qu'on pouvait fixer *ctratio* à 0.001 que nv dépendait du nombre de variables n du problème. Fixer nv avec la formule $nv = max(3, \frac{n}{4})$ a donné les meilleurs résultats. Les résultats expérimentaux ne sont pas mauvais, mais cette approche ne permet pas à `numVarCID` d'atteindre 0, c.-à-d. de n'appeler que la propagation de contraintes. C'est un inconvénient majeur quand une simple propagation est la méthode la plus efficace.

2.2 ACID1 : succession de phases d'apprentissage et d'exploitation

Une approche plus sophistiquée pallie cet inconvénient. `ACID1` alterne des phases d'apprentissage et d'exploitation pour l'auto-adaptation de la valeur *numVarCID*. Selon le numéro du nœud courant, l'algorithme est dans une phase d'apprentissage ou d'exploitation. Le comportement de `ACID1`, décrit par l'algorithme 2, est le suivant :

- Les variables sont triées selon leur impact (mesuré par la formule de l'heuristique `SmearSumRel`).
- Pendant une phase d'apprentissage (pendant *learnLength* nœuds), nous analysons l'évolution du ratio de contraction d'un appel à `var3BCID` au suivant et enregistrons le nombre *kvarCID* de rognages nécessaire pour obtenir l'essentiel du filtrage.
  Nous n'atteignons pas nécessairement le point-fixe en terme de filtrage à chaque nœud, et nous nous limitons à un nombre $2.numVarCID$ de variables rognées (avec un minimum égal à 2). Dans la première phase d'apprentissage, nous rognons n variables, c'est-à-dire que nous effectuons le premier appel à `ACID1` avec *numVarCID* $= 0.5\,n$.
  Pour le nœud courant, la fonction `lastSignificantGain` renvoie un nombre *kvarCID* de variables rognées, le $kvarCID^{eme}$ appel ayant donné la dernière contraction significative. Après cet appel à `var3BCID`, le gain sur la boîte courante produit par un appel à `var3BCID` et calculé par la formule `gainRatio`, n'excède jamais un ratio donné, appelé *ctratio*. Cette analyse commence par la dernière variable rognée. (Pour la lisibilité du pseudo-code, nous avons omis les paramètres de la procédure `var3BCID`, c.-à-.d. s_{3b}, s_{cid}, les contraintes C et le contracteur *ctc*.)
- Pendant la phase d'exploitation suivant une phase d'apprentissage, la moyenne des différentes valeurs *kvarCID* (obtenues dans les nœuds de la phase d'apprentissage) fournit la nouvelle valeur de *numVarCID*. Cette valeur sera utilisée pendant toute la phase d'exploitation. On notera que cette valeur peut au plus être le double de celle de la précédente phase d'exploitation, mais peut aussi significativement baisser.

Tous les `cycleLength` nœuds dans l'arbre de recherche, les 2 phases sont réappelées.

De nombreuses variantes de ce schéma ont été testées. En particulier, il est apparu contre-productif d'avoir une seule phase d'apprentissage et donc d'apprendre *numVarCID* une seule fois, ou au contraire de mémoriser les calculs d'une phase d'apprentissage à la suivante.

Nous avons fixé expérimentalement les 3 paramètres de la procédure `ACID1`, à savoir `learnLength`, `cycleLength` et `ctratio`, respectivement à 50, 1000 et

```
Procedure ACID1 (X, n, in-out [X], in-out call, in-out numVarCID)
    learnLength ← 50
    cycleLength ← 1000
    ctratio ← 0.002
    /* Tri des variables selon leur impact */
    X ← smearSumRelSort (X)
    if call % cycleLength ≤ learnLength then
        /* Phase d'apprentissage */
        nvarCID ← max(2, 2 . numVarCID)
        for i from 1 to nvarCID do
            [X]^old ← [X]
            var3BCID (X[i%n], [X], ...)
            ctcGains[i] ← gainRatio( [X], [X]^old)
        kvarCID[call] ← lastSignificantGain (ctcGains, ctratio, nvarCID)
        if call % cycleLength = learnLength then
            /* Fin de la phase d'apprentissage */
            numVarCID ← average (kvarCID[])
    else
        /* Phase d'exploitation */
        if numVarCID > 0 then
            for i from 1 to numVarCID do
                var3BCID (X[i % n], [X], ...)
    call ← call + 1
```

Algorithm 2: Algorithm ACID1

```
Function lastSignificantGain(ctcGains, ctratio, nvarCID)
    for i from nvarCID downto 1 do
        if (ctcGains[i] > ctratio) then
            return i
    return 0
```

0.002. ACID1 est donc devenue une procédure sans paramètres. Avec ces valeurs, le surcoût des phases d'apprentissage (pendant lesquelles la valeur précédente de *numVarCID* est doublée) reste faible.

2.3 ACID2 : Prise en compte de la profondeur dans l'arbre de recherche

Une critique peut être formulée envers ACID1 : la moyenne kvarCID est faite avec des valeurs obtenues à différentes profondeurs de l'arbre de recherche. Cet inconvénient est partiellement corrigé par les phases d'apprentissage successives de ACID1, où chaque phase correspond à une partie de l'arbre de recherche.

Pour aller plus loin, nous avons conçu un raffinement de ACID1 où chaque phase d'apprentissage règle plusieurs valeurs différentes suivant la largeur de la boîte étudiée. Une valeur correspond à un ordre de grandeur dans la largeur de la boîte. Par exemple, nous déterminons une valeur *numVarCID* pour les boîtes ayant une largeur comprise entre 1 et 0.1, une autre pour les boîtes avec une largeur entre 0.1 et 0.01, etc. Finalement, cette approche, appelée ACID2, a donné en général des résultats similaires à ceux de ACID1 et est apparu moins robuste. En effet, à certains niveaux de largeur de boîte, le réglage a été effectué sur quelques nœuds seulement, ce qui le rend peu significatif.

3 Expérimentations

Tous les algorithmes ont été implantés dans la bibliothèque logicielle de résolution par intervalles en C++ Ibex (Interval Based EXplorer) [6]. Les expérimentations ont été effectuées sur la même machine (Intel X86 3GHz). Nous avons testé les algorithmes pour la résolution de NCSP carrés et sur des problèmes d'optimisation globale sous contraintes. Résoudre un NCSP consiste à trouver toutes les solutions d'un système bien contraint de n équations non linéaires portant sur n variables réelles bornées. L'optimisation globale consiste à trouver le minimum global d'une fonction à n variables, sous des contraintes (équations et inégalités), la fonction objectif et/ou les contraintes étant non convexes.

3.1 Expérimentations en satisfaction de contraintes

Nous avons sélectionné dans le banc d'essai COPRIN [1] tous les systèmes qui pouvaient être résolus par un des algorithmes testés dans un temps compris entre 2 s et 3600 s. La limite de temps a été fixée à 10000 s. La précision requise pour une solution est 10^{-8}. Certains de ces problèmes peuvent avoir une taille (nombre de variables) à fixer. Dans ce cas, nous avons choisi la plus grande taille pouvant être résolue par un des algorithmes en moins d'une heure.

Nous avons comparé notre méthode ACID et ses variantes avec les techniques de filtrage bien connues : une simple propagation de contraintes HC4, 3BCID-n (voir partie 2) et 3BCID-fp (point fixe) pour laquelle on lance une nouvelle itération sur toutes les variables quand un domaine est réduit de plus d'1%. À chaque nœud de l'arbre de recherche, nous avons utilisé la séquence suivante de contracteurs : HC4, *shaving*, Interval-Newton [8] et X-Newton [2]. *shaving* dénote une variante de ACID, 3BCID-n, 3BCID-fp, ou rien si HC4 seul est testé.

1. www-sop.inria.fr/coprin/logiciels/ALIAS/Benches/benches.html

TABLE 1 – Résultats obtenus par ACID1 en résolution de CSP continus. Pour chaque problème, sont indiqués son nombre de variables et les résultats obtenus par ACID1 : le temps de calcul, le nombre de branchements, le nombre moyen de variables rognées (déterminé dynamiquement par ACID1). Nous indiquons aussi la meilleure et la plus mauvaise méthode parmi ACID1, HC4, 3BCID-fp, et 3BCID-n, le rapport en temps de calcul entre ACID1 et la meilleure méthode, et entre ACID1 et la pire méthode.

	#var	ACID1	ACID1	ACID1	meilleur	pire	ratio en temps	ratio en temps
		temps	#nœuds	#varcids			ACID1/meilleur	ACID1/pire
Bellido	9	3.45	518	5	ACID1	HC4	1	0.89
Brown-7	7	396	540730	4.5	ACID1	HC4	1	0.82
Brent-10	10	17.6	3104	9	ACID1	HC4	1	0.14
Butcher8a	8	981	204632	9	3BCID-n	HC4	1.03	0.49
Butcher8b	8	388	93600	10.8	ACID1	HC4	1	0.31
Design	9	29.2	5330	11	3BCID-n	HC4	1.07	0.37
Dietmaier	12	926	82364	26.3	ACID1	HC4	1	0.19
Directkin	11	32.7	2322	7	ACID1	3BCID-fp	1	0.84
Disc.integralf2-16	32	592	58464	0.4	HC4	3BCID-fp	1.02	0.52
Eco-12	11	3156	297116	12	ACID1	HC4	1	0.32
Fredtest	6	25.2	11480	0.8	HC4	3BCID-fp	1.04	0.91
Fourbar	4	437	183848	0.1	ACID1	3BCID-n	1	0.85
Geneig	6	178	83958	2.9	HC4	3BCID-fp	1.02	0.82
Hayes	7	3.96	1532	7.5	3BCID-n	HC4	1.14	0.77
I5	10	15.9	3168	11.5	ACID1	HC4	1	0.13
Katsura-25	26	691	5396	10.4	ACID1	3BCID-fp	1	0.67
Pramanik	3	23.1	23696	0.2	ACID1	HC4	1	0.69
Reactors-42	42	1285	23966	134	3BCID-fp	HC4	1.07	0.13
Reactors2-30	30	1220	38136	90	3BCID-n	HC4	1.14	0.12
Synthesis	33	356	7256	53.8	3BCID-fp	HC4	1.15	0.25
Trigexp2-23	23	2530	227136	39.4	3BCID-fp	HC4	1.26	0.25
Trigo1-18	18	2625	37756	6.1	ACID1	3BCID-fp	1	0.8
Trigo1sp-35	36	2657	70524	2.4	ACID1	3BCID-fp	1	0.41
Virasoro	8	1592	266394	0.6	3BCID-n	3BCID-fp	1.08	0.28
Yamamura1-16	16	2008	68284	0.4	3BCID-n	HC4	1.02	0.86
Yamamura1sp-500	501	1401	146	144	ACID1	HC4	1	0.14

Pour chaque problème, nous utilisons la meilleure heuristique de bissection parmi deux variantes de la fonction Smear [12]. Le paramètre principal *ctratio* de ACID1 et ACID2, mesurant une stagnation dans le contraction quand les variables sont rognées, a été fixé à 0.002. Les paramètres de var3BCID, s_{3b} and s_{cid}, ont été fixés à leurs valeurs par défaut, respectivement 10 et 1, proposées dans [13]. Des expérimentations sur les instances sélectionnées ont confirmé que ces valeurs étaient pertinentes et robustes. En particulier, fixer s_{3b} à 10 donne de meilleurs résultats que des valeurs plus petites ($s_{3b} = 5$) ou plus grandes ($s_{3b} = 20$). Pour 21 des 26 instances, $s_{3b} = 20$ a donné de plus mauvais résultats. Comme on peut le voir sur le tableau 1, ACID1 apparaît souvent comme le meilleur, ou proche du meilleur. Sur seulement 4 problèmes sur 26, il est plus de 10% plus lent que le meilleur. Le nombre de variables rognées a été réglé près de 0 dans les problèmes où HC4 suffisait, et à plus que le nombre de variables dans les problèmes où 3BCID-fp est apparue être la meilleure méthode.

Sur la partie gauche du tableau 2, on a résumé les résultats obtenus par les 3 variantes de ACID et leurs concurrents.

ACID1 est le seul qui a pu résoudre les 26 problèmes en 1 heure, tandis qu'HC4 n'a pu résoudre que 21 problèmes en 10000 s. Les gains en temps de calcul obtenus par ACID1 sur ses concurrents sont assez significatifs (voir la ligne *gain max*), tandis que ses pertes restent faibles. ACID0 avec ses deux paramètres a été plus difficile à régler, et les expérimentations n'ont pas montré d'intérêt à utiliser l'algorithme plus complexe ACID2. ACID1 obtient de meilleurs gains par rapport à 3BCID-n en temps total qu'en moyenne parce que les plus grands gains ont été obtenus sur des instances difficiles avec un grand nombre de variables. Sur la partie

TABLE 2 – NCSP : Gains en temps de calcul. Sont indiqués : le nombre de problèmes résolus en 3600 s et en 10000 s et différentes statistiques sur le temps CPU, ratio entre `ACID1` et chaque concurrent C_i (un par colonne) : la moyenne, le maximum, le minimum et l'écart type du rapport $\frac{ACID1\ time}{C_i\ time}$.

	`ACID1`	`HC4`	`3BCID-fp`	`3BCID-n`	`ACID0`	`ACID2`	`ACID1`	`3BCID-fp`	`3BCID-n`
							$\neg$ XN	$\neg$ XN	$\neg$ XN
#pbs résolus< 3600	26	20	23	24	25	24	20	16	20
#pbs résolus< 10000	26	21	26	26	26	26	22	21	22
Gain moyen	1	0.7	0.83	0.92	0.96	0.91	1	0.78	1.02
Gain maximum	1	0.13	0.26	0.58	0.45	0.48	1	0.18	0.38
Perte maximum	1	1.04	1.26	1.14	1.23	1.05	1	2.00	1.78
Ecart type gains	0	0.32	0.23	0.15	0.15	0.19	0	0.34	0.28
Temps total	23594	>72192	37494	27996	26380	30428	29075	50181	31273
Gain total	1		0.63	0.84	0.89	0.78	1	0.58	0.93

droite du tableau, on indique les rapports de temps de résolution obtenus quand on enlève `X-Newton` de la séquence des contracteurs (4 problèmes n'ont pas pu être résolus en 10000 s). La seule variante d'`ACID` étudiée est `ACID1`. `ACID1` et `3BCID-n` obtiennent globalement des résultats similaires, meilleurs que `3BCID-fp`, mais avec un plus grand écart type qu'avec `X-Newton`, car le rognage prend une part plus importante dans la contraction.

3.2 Expérimentations en optimisation globale sous contraintes

Nous avons sélectionné dans la série 1 du banc d'essais Coconut d'optimisation globale sous contraintes [2] les 40 instances qu'`ACID` ou un concurrent peut résoudre en un temps compris entre 2 s and 3600 s. La limite en temps de calcul a été fixée à 3600 s. Nous avons utilisé `IbexOpt`, la stratégie d'`Ibex` qui réalise un algorithme de Branch & Bound en meilleur d'abord. Le protocole expérimental est le même que pour la résolution des NCSP, sauf que nous n'avons pas utilisé `Interval-Newton`, qui n'est implanté que pour les systèmes carrés.

Pour chaque instance, nous avons utilisé la meilleure heuristique de bissection (la même pour toutes les méthodes) parmi `largestFirst`, `roundRobin` et des variantes de la fonction *Smear*. La précision requise sur l'objectif est 10^{-8}. Chaque équation est relâchée en 2 inégalités avec une précision égale à 10^{-8}.

Le tableau 3 a les mêmes colonnes que le tableau 1, avec en plus une colonne indiquant le nombre de contraintes de l'instance.

Pour ce qui concerne la programmation par contraintes dans `IbexOpt`, `HC4` correspond à l'état de l'art et `3BCID` est rarement utile [3]. C'est pourquoi nous présentons dans l'avant dernière colonne une comparaison entre `ACID1` et `HC4`. Le nombre de variables rognées a en effet été réglé par `ACID1` à une valeur comprise entre 0 et le nombre de variables. De nouveau, on peut remarquer qu'`ACID1` est robuste et est le meilleur, ou au plus 10% plus lent que le meilleur, pour 34 des 40 instances. Le tableau 4 montre que nous avons obtenu un gain moyen de 10% sur `HC4`. C'est significatif car la contraction due aux contraintes représente seulement une partie de la stratégie `IbexOpt` [12] (les relaxations linéaires et la recherche de points faisables sont d'autres algorithmes appartenant à la stratégie par défaut de `IbexOpt` qui ne sont pas étudiés dans cet article). `ACID0` rogne un minimum de 3 variables, ce qui est souvent trop. `ACID2` obtient des résultats légèrement moins bons qu'`ACID1`, ce qui rend en pratique ce raffinement non prometteur.

4 Conclusion

Nous avons présenté dans cet article une version adaptative de l'opérateur de contraction `3BCID` utilisé par des méthodes à intervalles, opérateur proche de `partition-1-AC` dans les CSP en domaines finis. La meilleure variante de cet opérateur adaptatif, appelé `ACID1` dans l'article, alterne des phases d'apprentissage et d'exploitation pour adapter le nombre de variables traitées. Ces variables sont sélectionnées par une heuristique de branchement efficace et tous les autres paramètres sont fixés et robustes aux modifications.

2. `www.mat.univie.ac.at/~neum/glopt/coconut/Benchmark/Benchmark.html`

3. En fait, l'algorithme récent de propagation de contraintes `Mohc` [1] est meilleur que `HC4`. `Mohc` n'est pas encore réimplanté en `Ibex` 2.0. Cependant, `3BCID(Mohc)` montre en gros les mêmes gains par rapport à `Mohc` que `3BCID(HC4)` par rapport à `HC4`...

`ACID1` n'ajoute aucun paramètre aux stratégies de résolution et d'optimisation. Il produit les meilleurs résultats en moyenne. Pour chaque instance traitée, il est le meilleur ou proche du meilleur, même en présence des autres contracteurs (`Interval-Newton`, `X-Newton`). C'est pourquoi nous l'avons ajouté aux stratégies par défaut de résolution et d'optimisation d'`Ibex`.

Références

[1] I. Araya, G. Trombettoni, and B. Neveu. Exploiting Monotonicity in Interval Constraint Propagation. In *Proc. AAAI*, pages 9–14, 2010.

[2] I. Araya, G. Trombettoni, and B. Neveu. A Contractor Based on Convex Interval Taylor. In *Proc. CPAIOR*, volume 7298 of *LNCS*, pages 1–16. Springer, 2012.

[3] F. Benhamou, F. Goualard, L. Granvilliers, and J.-F. Puget. Revising Hull and Box Consistency. In *Proc. ICLP*, volume 5649 of *LNCS*, pages 230–244. Springer, 1999.

[4] H. Bennaceur and M.-S. Affane. Partition-k-AC : An Efficient Filtering Technique Combining Domain Partition and Arc Consistency. In *Proc. CP*, volume 2239 of *LNCS*, pages 560–564. Springer, 2001.

[5] C. Bessiere and R. Debruyne. Optimal and Suboptimal Singleton Arc Consistency Algorithms. In *Proc. IJCAI*, pages 54–59, 2005.

[6] G. Chabert and L. Jaulin. Contractor Programming. *Artificial Intelligence*, 173 :1079–1100, 2009.

[7] R. Debruyne and C. Bessiere. Some Practicable Filtering Techniques for the Constraint Satisfaction Problem. In *Proc. IJCAI*, pages 412–417, 1997.

[8] E. Hansen. *Global Optimization using Interval Analysis*. Marcel Dekker inc., 1992.

[9] O. Lhomme. Consistency Techniques for Numeric CSPs. In *Proc. IJCAI*, pages 232–238, 1993.

[10] F. Messine. *Méthodes d'optimisation globale basées sur l'analyse d'intervalle pour la résolution des problèmes avec contraintes*. PhD thesis, LIMA-IRIT-ENSEEIHT-INPT, Toulouse, 1997.

[11] C. Min Li and Anbulagan. Heuristics Based on Unit Propagation for Satisfiability Problems. In *Proc. IJCAI*, pages 366–371, 1997.

[12] G. Trombettoni, I. Araya, B. Neveu, and G. Chabert. Inner Regions and Interval Linearizations for Global Optimization. In *Proc. AAAI*, pages 99–104, 2011.

[13] G. Trombettoni and G. Chabert. Constructive Interval Disjunction. In *Proc. CP*, volume 4741 of *LNCS*, pages 635–650. Springer, 2007.

TABLE 3 – Résultats obtenus par ACID1 en optimisation

	#var	#ctr	ACID1	ACID1	ACID1	meilleur	pire	Temps	Temps	Temps
			temps	#nœuds	#vcids			ACID1/meilleur	ACID1/HC4	ACID1/pire
Ex2_1_7	20	10	8.75	465	3	HC4	3BCID-fp	1.03	1.03	0.7
Ex2_1_8	24	10	6.18	200	0	HC4	3BCID-fp	1.06	1.06	0.91
Ex2_1_9	10	1	10.1	1922	0.75	HC4	3BCID-fp	1.04	1.04	0.9
Ex5_4_4	27	19	915	23213	0.8	ACID1	3BCID-n	1	0.96	0.91
Ex6_1_1	8	6	60.8	13071	8.9	HC4	3BCID-fp	1.21	1.21	0.73
Ex6_1_3	12	9	297	29154	11.7	HC4	3BCID-fp	1.19	1.19	0.63
Ex6_1_4	6	4	1.99	505	6	ACID1	3BCID-fp	1	0.97	0.8
Ex6_2_6	3	1	107	46687	0	HC4	3BCID-fp	1.02	1.02	0.74
Ex6_2_8	3	1	48.2	21793	0.1	HC4	3BCID-fp	1.01	1.01	0.72
Ex6_2_9	4	2	51.9	19517	0.1	HC4	3BCID-fp	1.02	1.02	0.72
Ex6_2_10	6	3	2248	569816	0	ACID1	3BCID-fp	1	0.99	0.64
Ex6_2_11	3	1	29.3	13853	0.3	HC4	3BCID-fp	1.05	1.05	0.73
Ex6_2_12	4	2	21.6	7855	0.1	HC4	3BCID-fp	1.02	1.02	0.8
Ex7_2_3	8	6	19.4	4596	4.4	3BCID-n	HC4	1.07	0.17	0.17
Ex7_2_4	8	4	36.8	5606	4.2	3BCID-fp	HC4	1.04	0.66	0.66
Ex7_2_8	8	4	38.0	6792	4.1	3BCID-n	HC4	1.09	0.71	0.71
Ex7_2_9	10	7	78.0	14280	9.3	3BCID-n	HC4	1.07	0.48	0.48
Ex7_3_4	12	17	2.95	366	3	3BCID-n	3BCID-fp	1.23	0.99	0.89
Ex7_3_5	13	15	4.59	894	6	3BCID-n	HC4	1.05	0.38	0.38
Ex8_4_4	17	12	1738	46082	0.9	ACID1	3BCID-fp	1	0.99	0.87
Ex8_4_5	15	11	772	25454	4.8	HC4	3BCID-fp	1.03	1.03	0.75
Ex8_5_1	6	5	9.67	2138	2.75	ACID1	3BCID-fp	1	0.84	0.82
Ex8_5_2	6	4	32.5	5693	0.8	ACID1	3BCID-fp	1	0.9	0.87
Ex8_5_6	6	4	32.4	10790	1.8	HC4	3BCID-fp	1.02	1.02	0.76
Ex14_1_7	10	17	665	95891	3.3	3BCID-n	HC4	1.03	0.61	0.61
Ex14_2_3	6	9	2.01	360	2	HC4	3BCID-fp	1.17	1.17	0.69
Ex14_2_7	6	9	49.9	5527	0	HC4	3BCID-n	1.47	1.47	0.48
alkyl	14	7	3.95	714	4	HC4	3BCID-fp	1.2	1.2	0.91
bearing	13	12	11.6	1098	13	3BCID-n	HC4	1.01	0.53	0.53
hhfair	28	25	26.6	3151	10	3BCID-n	HC4	1.12	0.58	0.58
himmel16	18	21	188	21227	15.5	3BCID-n	3BCID-fp	1.1	0.94	0.88
house	8	8	62.8	27195	3.25	HC4	3BCID-fp	1.09	1.09	0.79
hydro	30	24	609	32933	0	ACID1	3BCID-fp	1	0.88	0.78
immun	21	7	4.17	1317	2.5	ACID1	3BCID-fp	1	0.55	0.28
launch	38	28	107	2516	21	ACID1	3BCID-n	1	0.79	0.43
linear	24	20	751	27665	0.25	ACID1	3BCID-n	1	0.98	0.65
meanvar	7	2	2.43	370	2	HC4	3BCID-fp	1.04	1.04	0.84
process	10	7	2.61	611	8	HC4	3BCID-fp	1.08	1.08	0.77
ramsey	31	22	164	4658	4.3	ACID1	3BCID-fp	1	0.85	0.68
srcpm	38	27	160	6908	0.5	ACID1	3BCID-fp	1	0.62	0.33

TABLE 4 – Problèmes d'optimisation : rapport des gains en temps de résolution : temps ACID1/temps xxx

	ACID1	HC4	3BCID-fp	3BCID-n	ACID0	ACID2
# pbs résolus	40	40	40	40	40	40
Gain moyen	1	0.9	0.77	0.88	0.91	0.97
Gain maximum	1	0.17	0.28	0.35	0.62	0.28
Perte maximum	1	1.47	1.04	1.23	1.18	1.19
Ecart type gains	0	0.25	0.16	0.18	0.12	0.14
Temps total	9380	10289	12950	11884	11201	9646
Gain total	1	0.91	0.72	0.79	0.84	0.97

Acquisition de contraintes avec des requêtes partielles

Christian Bessiere[1] **Remi Coletta**[1] **Emmanuel Hebrard**[2] **George Katsirelos**[3]
Nadjib Lazaar[1] **Nina Narodytska**[4] **Claude-Guy Quimper**[5] **Toby Walsh**[4]

[1]CNRS, Université de Montpellier, France
[2]LAAS-CNRS, Toulouse, France
[3]INRA Toulouse, France
[4]NICTA, UNSW, Sydney, Australie
[5]Université Laval, Québec, Canada

{bessiere,coletta,lazaar}@lirmm.fr hebrard@laas.fr gkatsi@gmail.com
ninan@cse.unsw.edu.au claude-guy.quimper@ift.ulaval.ca toby.walsh@nicta.com.au

Résumé

Nous apprenons des réseaux de contraintes en utilisant des requêtes partielles. Autrement dit, nous demandons à l'utilisateur de classer une affectation de sous-ensembles de variables comme positive ou négative. Nous fournissons un algorithme qui, étant donné un exemple complet négatif, apprend une contrainte du réseau cible avec un certain nombre de requêtes partielles, logarithmique en taille de l'exemple. Nous présentons une étude théorique sur les bornes inférieures en termes de requêtes pour apprendre certaines classes de réseaux de contraintes et montrons l'optimalité de notre algorithme générique dans certains cas. Enfin, nous évaluons expérimentalement notre algorithme.

Abstract

We learn constraint networks by asking the user partial queries. That is, we ask the user to classify assignments to subsets of the variables as positive or negative. We provide an algorithm that, given a negative example, focuses onto a constraint of the target network in a number of queries logarithmic in the size of the example. We give information theoretic lower bounds for learning some simple classes of constraint networks and show that our generic algorithm is optimal in some cases. Finally we evaluate our algorithm on some benchmarks.

1 Introduction

La programmation par contraintes (PPC) connait un succès croissant et est largement utilisée dans de nombreuses applications industrielles. Toutefois, la première difficulté que rencontre un utilisateur de ce paradigme est la modélisation. Quelle est la traduction fidèle de la spécification du problème en contraintes ? Plusieurs techniques ont été proposées pour répondre à ce besoin de façon automatique. Freuder et Wallace ont proposé dans [7] des stratégies de suggestion (*matchmaker*) pour les applications où les utilisateurs n'ont pas prévu toutes les contraintes à l'avance mais sont néanmoins capables d'en proposer à chaque fois que le système retourne une non-solution. La première version de Conacq (Conacq.1) [3, 4] implémente une acquisition *passive* de contraintes où l'utilisateur fournit un ensemble de solutions/non-solutions (ex. des emplois du temps conçus à la main). Sur la base de ses exemples, Conacq.1 apprend un ensemble de contraintes qui accepte/rejette toutes les solutions/non-solutions fournies au départ. Dans le même cadre passif, une autre approche utilise des solutions et non-solutions et réalise l'acquisition des contraintes grâce à une base de connaissances (incluant des interprétations logiques des solutions/non-solutions) et des techniques de Programmation Logique Inductive [10]. Dans [2], Beldiceanu et Simonis ont proposé ModelSeeker, un système d'acquisition de contraintes passive basé sur le catalogue des contraintes globales, dont la portée des contraintes sont des lignes, des colonnes ou toute autre propriété structurelle que peut capturer ModelSeeker. Seuls les exemples positifs (solutions) sont fournis par l'utilisateur. ModelSeeker implémente également une

technique efficace qui permet de classer les meilleures contraintes candidates sur un ensemble particulier de variables.

La version *active* de CONACQ (CONACQ.2) [5] sollicite l'aide de l'utilisateur à classifier des exemples comme positives/négatives (solutions/non-solutions) avec des requêtes d'appartenance complètes (c.-à-d. affectation complète des variables) [1]. Une acquisition active des contraintes présente plusieurs avantages : i) Le nombre de requêtes nécessaires pour converger à l'ensemble de contraintes cible peut être considérablement réduit. ii) l'utilisateur peut être une machine (ex. système expert, simulateur, etc.). Par exemple, la société NORMIND a récemment recruté un spécialiste en PPC pour transformer un système expert qui détecte les défaillances dans les circuit électroniques d'un avion Airbus en un modèle à contraintes, afin de rendre l'outil plus efficace et facile à maintenir. Un autre exemple est celui d'apprendre les contraintes qui traduisent les actions non atomiques d'un robot (ex. attraper une balle) en posant des requêtes à un simulateur de robot [11]. Le cadre actif d'acquisition de contraintes évoque deux défis connus en apprentissage automatique : i) La qualité et comment générer des requêtes ? ii) Le nombre de requêtes nécessaires pour converger vers l'ensemble des contraintes cible ? Concernant le deuxième point, il a été démontré que le nombre de requêtes complètes nécessaires pour atteindre le réseau de contraintes cible est exponentiel dans le cas général.

Dans cet article, nous proposons QUACQ (pour *QuickAcquisition*), un système actif d'acquisition de contraintes qui demande à l'utilisateur de classifier, non seulement des requêtes complètes, mais aussi des requêtes *partielles*. Étant donné un exemple négatif, QUACQ est capable d'apprendre une contrainte du réseau cible avec un nombre de requêtes (partielles) logarithmiques en taille de l'exemple. Nous présentons également une étude théorique sur les bornes inférieures en termes de requêtes pour apprendre certaines classes de réseaux de contraintes et montrons l'optimalité de notre algorithme générique dans certains cas. Enfin, nous évaluons expérimentalement notre algorithme.

QUACQ est défini de sorte à pouvoir apprendre un modèle générique. En PPC, les données sont généralement séparées du modèle. Le modèle du Sudoku, par exemple, contient des contraintes génériques comme la permutation des valeurs dans chaque carré. D'autre part, les données définissent les différentes grilles de sudoku avec les cases pré-remplies. Autre exemple, le problème d'emploi du temps, le modèle contient des contraintes génériques comme par exemple « aucun enseignant ne peut donner deux cours dans une même plage horaire ». Par contre, les données permettent de préciser le nombre des salles, la disponibilité d'un enseignant donné pour un créneau donnée, etc. Un des avantages de cette approche est qu'elle permet de réduire l'effort de l'utilisateur. Premièrement, QUACQ converge plus rapidement que les autres systèmes. Deuxièmement, les requêtes partielles sont plus faciles à classifier que les requêtes complètes. Troisièmement, contrairement aux autres approches, QUACQ n'a pas besoin d'exemples positifs pour apprendre l'ensemble des contraintes : un point intéressant lorsque le problème en question n'a jamais été résolu.

2 Contexte

L'apprenant et l'utilisateur doivent partager un même *vocabulaire* pour pouvoir communiquer. Dans le cadre d'acquisition de contraintes, ce vocabulaire est défini par un ensemble (fini) de variables X et leurs domaines $D = \{D(X_i)\}_{X_i \in X}$, où $D(X_i) \subset \mathbb{Z}$ est le domaine fini de la variable X_i. Étant donnée une suite de variables $S \subseteq X$, une *contrainte* représente une paire (c, S) (notée C_S), où c est une relation sur $\mathbb{Z}$ précisant les tuples autorisés pour les variables S. S étant la *portée* de la contrainte C_S. Un *réseau de contraintes* est un ensemble C de contraintes sur le vocabulaire (X, D). Une affectation e_Y sur un sous-ensemble de variables $Y \subseteq X$ est *rejetée* par une contrainte c_S si $S \subseteq Y$ et la projection $e_Y[S]$ de e sur les variables de S n'est pas dans c_S. Une affectation complète sur X est une *solution* de C *ssi* elle n'est rejetée par aucune contrainte dans C. Nous utilisons la notation $sol(C)$ pour l'ensemble des solutions de C et la notation $C[Y]$ pour l'ensemble des contraintes de C dont la portée est incluse dans Y.

En terme d'apprentissage automatique, nous utilisons la notion de *biais de contraintes*, noté B, qui représente un ensemble de contraintes construit à partir d'un langage Γ sur le vocabulaire (X, D). C'est à partir du biais B que l'apprenant construit le réseau de contraintes. Un *concept* est une fonction booléenne sur $D^X = \Pi_{X_i \in X} D(X_i)$. Etant donné un exemple $e \in D^X$, un *concept cible* f_T renvoie 1 pour e *ssi* e est une solution du problème que l'utilisateur a en tête, 0 autrement. Une *requête d'appartenance* $ASK(e)$ est une question de classification posée à l'utilisateur, où e est une affectation complète sur D^X. La réponse à $ASK(e)$ étant *oui ssi* $f_T(e) = 1$.

Pour être en mesure d'utiliser des requêtes *partielles*, nous posons une condition supplémentaire sur les capacités de l'utilisateur. Même si ce dernier n'est pas en mesure d'exprimer les contraintes de son problème, il est capable de décider si une instantiation partielle des variables X viole certaines exigences ou non. Un *réseau cible* est un réseau C_T tel que $sol(C_T) = \{e \in$

$D^X \mid f_T(e) = 1\}$. Une requête *partielle* $ASK(e_Y)$, avec $Y \subseteq X$, est une question de classification posée à l'utilisateur, où e_Y est une affectation *partielle* dans $D^Y = \Pi_{X_i \in Y} D(X_i)$. Un ensemble de contraintes C *accepte* une requête partielle e_Y *ssi* il n'existe aucune contrainte c_S dans C qui rejette $e_Y[S]$. La réponse à $ASK(e_Y)$ étant *oui ssi* C_T accepte e_Y. Pour toute affectation e_Y sur Y, nous notons $\kappa_B(e_Y)$ le sous-ensemble des contraintes de B violées par e_Y. Une affectation e_Y est dite *positive* ou *négative* selon la classification de l'utilisateur.

Nous nous intéressons au problème de *convergence* en acquisition de contraintes. Etant donné un ensemble E d'exemples (partiels) classés par l'utilisateur (en positif/negatif), on dit qu'un réseau de contraintes C est en conformité avec E si C accepte tous les positifs et rejette tous les négatifs dans E. On dit que le processus d'acquisition a *convergé* sur le réseau $C_L \subseteq B$ si C_L est en conformité avec E et pour tout autre réseau $C' \subseteq B$ en conformité avec E, nous avons $sol(C') = sol(C_L)$. S'il n'existe pas de $C_L \subseteq B$ tel que C_L est en conformité avec E, nous disons que nous avons atteint un état *d'effondrement* . Cela se produit lorsque $C_T \not\subseteq B$.

3 Algorithme Générique d'Acquisition de Contraintes

Dans cette section, nous présentons QuAcq, une nouvelle approche d'acquisition active de contraintes. QuAcq prend en entrée un biais B sur un vocabulaire (X, D). Il génère des requêtes (partielles) que pose à l'utilisateur jusqu'à ce qu'il atteigne un état de convergence en retournant un réseau de contraintes C_L équivalent au réseau cible C_T, ou un état d'effondrement. Lorsqu'un utilisateur répond par *oui* à une requête complète, on retire du biais B les contraintes violées par cet exemple. Dans le cas d'un *non*, qa entre dans une boucle (fonctions `FindScope` et `FindC`) qui permet d'ajouter d'une contrainte à C_L.

3.1 Description de QuAcq

qa (algo. 1) initialise le réseau qu'il va apprendre C_L à l'ensemble vide (ligne 1). Si C_L est unsatisfiable (ligne refqa :fail), l'espace des réseaux possibles s'effondre parce qu'il n'existe plus de sous-ensemble du biais B capable de classer correctement les requêtes déjà posées. A la ligne 4, QuAcq calcule une affectation complète e satisfaisant le C_L courant et viole au moins une contrainte du biais B. Si un tel exemple n'existe pas (ligne 5), alors toutes les contraintes qui restent dans B sont des contraintes impliquées par C_L, ce qui nous conduit à un état de convergence. Autrement, QuAcq pose la question $ASK(e)$ à l'utilisateur, qui pourra répondre par *oui* ou *non*.

Si la réponse est *oui*, nous pouvons retirer de B l'ensemble $\kappa_B(e)$ des contraintes qui rejettent l'exemple e (ligne6). Dans le cas d'un *non*, cela veut dire que e viole au moins une contrainte du réseau cible C_T. Nous appelons ensuite la fonction `FindScope` pour calculer la portée d'au moins une contrainte violée par e. La fonction `FindC` permet par la suite de déterminer quelle contrainte, avec une telle portée, a été violée par e (ligne 8). Si aucune contrainte n'est retournée par `FindC` (ligne 9), c'est encore une condition suffisante pour atteindre un état d'effondrement (c.-à-d. aucune contrainte de B ne rejette l'exemple négatif e). Autrement, la contrainte retournée par `FindC` est ajoutée au réseau C_L (ligne 10).

Algorithm 1: QuAcq : Acquisition de contraintes avec des requêtes partielles

```
1  C_L ← ∅;
2  while true do
3      if sol(C_L) = ∅ then return "collapse";
4      choose e in D^X accepted by C_L and rejected
       by B ;
5      if e = nil then return "convergence on C_L";
6      if ASK(e) = yes then B ← B \ κ_B(e) ;
7      else
8          c ← FindC(e, FindScope(e, ∅, X, false));
9          if c = nil then return "collapse";
10         else C_L ← C_L ∪ {c};
```

La fonction récursive `FindScope` prend en paramètres un exemple négatif e, deux ensembles de variables R et Y, et un booléen ask_query. Un invariant de `FindScope` est que e viole au moins une contrainte dont la portée est un sous-ensemble de $R \cup Y$. Lorsque `FindScope` est appelée avec ask_qery = **false**, nous savons déjà si R contient la portée d'une contrainte qui rejette e (ligne 1). Si ask_qery = **true**, nous demandons à l'utilisateur si la projection $e[R]$ est positive ou non (ligne 2). Si *oui*, nous pouvons retirer de B les contraintes qui rejettent $e[R]$, sinon, nous retournons l'ensemble vide (ligne 4). La ligne 5 est atteinte uniquement dans le cas où $e[R]$ ne viole aucune contrainte. Il est évident que $e[R \cup Y]$ viole au moins une contrainte. Par conséquent, et sachant que Y est un singleton, la variable dans Y appartient nécessairement à la portée d'une contrainte violée par $e[R \cup Y]$. La fonction retourne donc Y. Si aucune des conditions de retour n'est satisfaite, l'ensemble Y est divisé en deux (ligne 6) et nous appliquons une technique similaire à QuickXplain [1][9] pour élucider les variables

1. La différence principale est que QuAcq divise un en-

Algorithm 2: Fonction `FindScope` : retourne la portée d'une contrainte dans C_T

```
   function FindScope(in e, R, Y, ask_query) : scope;
   begin
1      if ask_query then
2          if ASK(e[R]) = yes then
3              B ← B \ κ_B(e[R]);
4          else return ∅;
5      if |Y| = 1 then return Y;
6      split Y into < Y1, Y2 > such that |Y1| = ⌈|Y|/2⌉ ;
7      S1 ← FindScope(e, R ∪ Y1, Y2, true);
8      S2 ← FindScope(e, R ∪ S1, Y1, (S1 ≠ ∅));
9      return S1 ∪ S2;
```

Algorithm 3: Fonction `FindC` : retourne une contrainte dans C_T avec une portée Y

```
   function FindC(in e, Y) : constraint;
   begin
1      B ← B \ {c_Y | C_L ⊨ c_Y};
2      Δ ← {c_Y ∈ B[Y] | e ⊭ c_Y};
3      if Δ = ∅ then return ∅;
4      while true do
5          choose e' in sol(C_L[Y]) such that
           ∃c, c' ∈ Δ, e' ⊨ c and e' ⊭ c';
6          if e' = nil then
7              if Δ = ∅ then return nil;
8              else pick c in Δ ; return c;
9          if ASK(e') = yes then
10             B ← B \ κ_B(e') ; Δ ← Δ \ κ_B(e');
11     else Δ ← Δ ∩ κ_B(e');
```

d'une contrainte violée par $e[R \cup Y]$ en un nombre logarithmique d'étapes (lignes (lines 7–9).

La fonction `FindC` prend en paramètres e et Y, e étant l'exemple négatif qui a permis à `FindScope` de retourner la portée Y d'une contrainte violée. `FindC` retire de B dans un premier temps toutes les contraintes de portée Y qui sont impliquées par le C_L courant (ligne 1) [2] L'ensemble Δ contient au départ l'ensemble des contraintes violées par e (ligne 2). Si Δ ne contient plus de contraintes avec une portée Y (ligne 3), `FindC` retourne $\varnothing$, ce qui provoquera un effondrement dans QuAcq. A la ligne 5, un exemple e' est généré de manière à avoir dans Δ à la fois des contraintes qui rejettent e' et d'autres qui l'acceptent. Si aucun exemple de ce type n'existe (ligne 6), cela signifie que l'ensemble des contraintes qui restent dans Δ est équivalent à $C_L[Y]$. Dans ce cas, `FindC` retourne une contrainte de Δ, sauf si Δ est vide (lignes 7-8). Si un exemple a été trouvé, on demande à l'utilisateur de le classifier (ligne 9). Si l'exemple est positif, les contraintes violées par cet exemple sont retirées de B et de Δ (ligne 10). Si l'exemple est négatif, on retire de Δ toutes les contraintes qui acceptent cet exemple. nous retirons de Δ toutes les contraintes accepter cet exemple (ligne 11).

3.2 Exemple Illustratif

Nous illustrons le fonctionnement de QuAcq sur un exemple simple. Considérons l'ensemble des variables $X_1, \ldots, X_5$ avec les domaines $\{1..5\}$, un langage $\Gamma = \{=, \neq\}$, un biais $B = \{=_{ij}, \neq_{ij} | \ i, j \in 1..5, i < j\}$, et un réseau de contraintes cible $C_T = \{=_{15}, \neq_{34}\}$. Supposons que le premier exemple généré à la ligne 4 dans QuAcq est $e_1 = (1, 1, 1, 1, 1)$. La trace d'exécution de `FindScope`$(e_1, \varnothing, X_1 \ldots X_5, \textbf{false})$ est présenté dans le tableau ci-dessous. Chaque ligne correspond à un appel à `FindScope`. Les requêtes portent toujours sur un sous-ensemble de variables de R. '$\times$' dans la colonne ASK signifie que l'appel précédent retourné un $\varnothing$, donc la question est ignorée. Les requêtes des lignes 1 et 2.1 dans le tableau ont permis à la fonction `FindScope` de retirer les contraintes $\neq_{12}, \neq_{13}, \neq_{23}$ et $\neq_{14}, \neq_{24}$ de B. Une fois la portée (X_3, X_4) est retourné, `FindC` exige a besoin d'un seul exemple pour retourner $\neq_{34}$ et retirer$=_{34}$ de B. Supposons maintenant que l'exemple suivant généré par QuAcq soit $e_2 = (1, 2, 3, 4, 5)$. `FindScope` retournera la portée (X_1, X_5) et `FindC` retournera par la suite $=_{15}$ le même traitement effectué sur e_1. Les contraintes $=_{12}, =_{13}, =_{14}, =_{23}, =_{24}$ sont retirées de B suite à une requête partielle positive sur $X_1, \ldots, X_4$ et $\neq_{15}$ par `FindC`. Ensuite, des exemples comme $e_3 = (1, 1, 1, 2, 1)$ et $e_4 = (3, 2, 2, 3, 3)$, positifs, permettront de retirer les contraintes $\neq_{25}, \neq_{35}, =_{45}$ et $=_{25}, =_{35}, \neq_{45}$ de B et atteindre la convergence.

call	R	Y	ASK	*return*
0	$\varnothing$	X_1, X_2, X_3, X_4, X_5	$\times$	X_3, X_4
1	X_1, X_2, X_3	X_4, X_5	yes	X_4
1.1	X_1, X_2, X_3, X_4	X_5	no	$\varnothing$
1.2	X_1, X_2, X_3	X_4	$\times$	X_4
2	X_4	X_1, X_2, X_3	yes	X_3
2.1	X_4, X_1, X_2	X_3	yes	X_3
2.2	X_4, X_3	X_1, X_2	no	$\varnothing$

semble de variables alors que QuickXplain divise un ensemble de contraintes

2. Cette étape pouvait être effectuée dans QuAcq, juste après la ligne 10, le coût de calcul étant réduit nous avons préféré l'effectuer dans `FindC`.

3.3 Complexité

Nous analysons la complexité de QuAcq en termes de nombre de requêtes. Les requêtes sont posées à l'utilisateur dans les lignes 6 de QuAcq, 2 de `FindScope` et 9 de `FindC`.

Proposition 1. *Etant donnés un biais B construit à partir d'un langage Γ, un réseau cible C_T et une portée Y, `FindC` a besoin de $O(|\Gamma|)$ requêtes pour retourner une contrainte c_Y de C_T si cette dernière existe.*

Démonstration. A chaque fois que `FindC` génère une requête, quelle que soit la réponse de l'utilisateur, la taille de l'ensemble Δ diminue strictement. Ainsi, le nombre total de requêtes posées dans `FindC` est majorée par $|\Delta|$, qui lui-même, est majorée par le nombre de contraintes du langage Γ d'arité $|Y|$ (voir ligne 2 de `FindC`).

□

Proposition 2. *Étant donnés un biais B, un réseau cible C_T et un exemple $e \in D^X \setminus sol(C_T)$, `FindScope` a besoin de $O(|S| \cdot \log|X|)$ requêtes pour retourner la portée S d'une des contraintes de C_T violée par e.*

Démonstration. `FindScope` est un algorithme récursif qui génère, au plus, une requête par appel (ligne 2). Par conséquent, le nombre de requêtes est majoré par le nombre de n ?uds de l'arbre des appels récursifs à `FindScope`. Nous allons montrer qu'une feuille de l'arbre est soit sur une branche qui conduit à une variable de la portée S qui sera retournée, ou elle représente le n ?ud fils d'une telle branche. Quand une branche ne conduit pas à une variable de la portée S, cette branche nous conduit nécessairement vers des feuilles qui correspondent à des appels de `FindScope` qui retournent $\varnothing$. La seule façon pour qu'un appel de `FindScope` au niveau des feuilles retourne l'ensemble vide est d'avoir reçu la réponse *non* à sa requête (ligne 4). Soit R_{fils}, Y_{fils} les valeurs des paramètres R et Y pour un appel feuille avec la réponse *non*, et R_{parent}, Y_{parent} les valeurs des paramètres R et Y de l'appel parent dans l'arborescence des appels récursifs. A partir de la réponse *non* sur la requête $ASK(e[R_{fils}])$, on peut déduire que $S \subseteq R_{fils}$ mais $S \nsubseteq R_{parent}$ parce que l'appel au niveau parent à reçu un *oui*. Considérons d'abord le cas où la feuille représente le fils gauche du n ?ud parent. Par construction, $R_{parent} \subsetneq R_{fils} \subsetneq R_{parent} \cup Y_{parent}$. En conséquence, Y_{parent} intersecte S, et le n ?ud parent est sur une branche qui conduit à une variable dans S. Considérons maintenant le cas où la feuille représente le fils droit du n ?ud parent. Comme nous sommes sur une feuille, si la variable ask_query est à faux, on sort nécessairement de `FindScope` par la ligne 5, ce qui signifie que ce n ?ud est la fin d'une branche qui nous conduit à une variable dans S. Ainsi, nous sommes sûrs d'avoir ask_query à vrai, ce qui signifie que le fils gauche du n ?ud parent a retourné un ensemble non vide et que le n ?ud parent est sur une branche qui mène vers une variable dans S.

Nous avons prouvé que chaque feuille est soit sur une branche qui mène vers une variable dans S, soit elle représente un fils d'un n ?ud qui mène vers une variable dans S. Ainsi, le nombre de n ?uds dans l'arbre est au plus deux fois le nombre de n ?uds dans les branches qui conduisent à une variable de S. Les branches peuvent être, au plus, de longueur $\log|X|$. Par conséquent, le nombre total de requêtes générées par `FindScope` est au plus $2 \cdot |S| \cdot \log|X|$, qui est dans $O(|S| \cdot \log|X|)$.

□

Theorem 1. *Etant donnés un biais B construit à partir d'un langage Γ de contraintes d'arité bornée, et un réseau cible C_T,* QuAcq *a besoin de $O(|C_T| \cdot (\log|X| + |\Gamma|))$ requêtes pour apprendre un réseau équivalent à C_T ou à s'effondrer, et de $O(|B|)$ requêtes pour prouver la convergence.*

Démonstration. A chaque fois qu'on qu'un exemple est classé négatif à la ligne 6 dans QuAcq, la portée d'une contrainte c_S dans C_T est retournée avec au plus $|S| \cdot \log|X|$ requêtes (proposition 2). Sachant que le langage Γ ne contient que des contraintes d'arité bornée, $|S|$ est également bornée et c_S est retournée avec $O(|\Gamma|)$ requêtes ou on atteint un état d'effondrement (proposition 1). Par conséquent, le nombre de requêtes nécessaires pour apprendre C_T ou de s'effondrer est en $O(|C_T| \cdot (\log|X| + |\Gamma|))$. La convergence est atteinte une fois B est réduit à vide grâce aux exemples classés positive à la ligne 6 dans QuAcq. Chacun de ces exemples conduit nécessairement au retrait d'au moins une contrainte de B suite à la façon dont on génère les exemples à la ligne 4. Cela donne un total de $O(|B|)$.

□

4 Langages Simples

Nous considérons des langages simples dans le but d'obtenir une estimation théorique de l'efficacité de QuAcq. Pour chacun de ces langages, nous analysons le nombre de requêtes nécessaires pour apprendre un réseau dans ce langage. Dans certains cas, nous montrons que QuAcq apprend les problèmes dans le langage donné avec un nombre de requêtes asymptotiquement optimal. Cependant, pour certains langages, un nombre de requêtes plus important est nécessaire dans le pire des cas. Dans notre analyse, nous considérons

le cas où la solution de C_L qui maximise le nombre de contraintes violées dans le biais B est choisie lorsqu'un exemple complet doit être généré (ligne 4 de QuAcq).

4.1 Langages pour lesquels QuAcq est optimal

Theorem 2. QuAcq *apprend un réseau booléen dans le langage* $\{=, \neq\}$ *avec un nombre de requêtes asymptotiquement optimal.*

Démonstration. (Esquisse.) Tout d'abord, nous donnons une borne inférieure sur le nombre de requêtes nécessaires pour apprendre un réseau dans ce langage. Prenons la restriction aux seules égalités. Dans une instance de ce langage, toutes les variables d'une composante connexe doivent être égales. L'ensemble de ces instances est donc isomorphique à l'ensemble des partitions de n objets, dont la taille est donnée par le *nombre de Bell* :

$$C(n+1) = \begin{cases} 1 & \text{si } n = 0 \\ \sum_{i=1}^{n} \binom{n}{i} C(n-i) & \text{si } n > 0 \end{cases} \quad (1)$$

Par un argument de la théorie de l'information, on peut donc déduire qu'au moins $\log C(n)$ requêtes sont nécessaires pour apprendre un tel problème. Celà implique une borne inférieure de $\Omega(n \log n)$, puisque $\log C(n) \in \Omega(n \log n)$ (voir [6] pour une preuve). Le langage $\{=, \neq\}$ est plus riche, et donc requiert au moins autant de requêtes.

Ensuite, nous considérons la requête soumise à l'utilisateur dans la ligne 6 de QuAcq, et nous faisons le compte des réponses *oui* et *non*. L'observation clé est qu'une instance de ce langage contient au plus $O(n)$ contraintes non redondantes. Pour chaque réponse *non* en ligne 6 de QuAcq, une nouvelle contrainte va être ajoutée à C_L. Seules les contraintes non redondantes sont découvertes de cette manière puisque la requête doit satisfaire C_L. Il s'en suit qu'au plus $O(n)$ telles requêtes reçoivent une réponse *non*, chacune entrainant $O(\log n)$ requêtes supplémentaires au travers de la procédure `FindScope`.

Maintenant nous pouvons borner le nombre de réponses *oui* à la ligne 6 de QuAcq. La même observation sur la structure de ce langage est à nouveau utile. Dans la version complète de la preuve, nous montrons que la requête qui maximise le nombre de contraintes violées dans le biais B tout en satisfaisant les contraintes de C_L viole au moins $\lceil |B|/2 \rceil$ contraintes de B. Donc, chaque requête dont la réponse est *oui* réduit de moitié le nombre de contraintes dans B. Il s'en suit que la requête soumise à la ligne 6 de QuAcq ne peut recevoir plus de $O(\log n)$ réponse *oui*. Le nombre total de requêtes est donc borné par $O(n \log n)$. □

Le même argument s'applique aux sous-langages $\{=\}$ et $\{\neq\}$ sur des domaines booléens. De plus, celà est toujours vrai pour $\{=\}$ sur des domaines arbitraires.

Corollary 1. QuAcq *apprend un réseau sur des domaines non bornés dans le langage* $\{=\}$ *avec un nombre de requêtes asymptotiquement optimal.*

4.2 Langages pour lesquels QuAcq est non-optimal

Tout d'abord, nous montrons qu'un réseau de contraintes booléennes dans le langage $\{<\}$ peut être appris avec $O(n)$ requêtes. Ensuite, nous montrons que QuAcq a besoin de $\Omega(n \log n)$ requêtes.

Theorem 3. *Les réseaux de contraintes booléennes dans le langage* $\{<\}$ *peuvent être appris en* $O(n)$ *requêtes.*

Démonstration. Notez que, pour décrire un tel problème, les variables peuvent être partitionnées en trois ensembles. i) Les variables qui doivent prendre la valeur 0 (c.-à-d. la partie gauche d'une contrainte $<$), ii) les variables qui doivent prendre la valeur 1 (c.-à-d. la partie droite d'une contrainte $<$), iii) et les variables libres (c.-à-d. les variables qui n'apparaissent dans aucune contrainte). Dans une première étape, on partitionne les variables en trois ensembles, G, D, I initialement vide et correspondant respectivement à *Gauche*, *Droite* et *Inconnu*. Durant cette étape, nous avons trois invariants :

1. Il n'existe pas de $x, y \in I$ tel que $x < y$ appartient au réseau cible C_T
2. $x \in G$ *ssi* il existe $y \in I$ et $x < y$ est dans le réseau cible C_T
3. $x \in D$ *ssi* il existe $y \in I$ et $y < x$ dans le réseau cible C_T

Nous passons par toutes les variables du problème, une à la fois. Soit x la dernière variable tirée. Nous posons une requête à l'utilisateur où x, ainsi que toutes les variables dans I sont mises à 0, et toutes les variables de D sont mises à 1 (variables dans G restent sans affectation). Si la réponse est *oui*, alors il n'y a pas de contrainte entre x et une variable $y \in I$, donc x est mise dans I sans toucher aux invariants. Autrement, x est soit impliquée dans une contrainte $y < x$ ou $x < y$ avec $y \in I$. Afin de décider quelle valeur prendre pour x, une deuxième requête est générée où la valeur de x passe à 1 et toutes les valeurs des autres variables restent inchangées. Si la réponse à cette requête est *oui*, alors la première hypothèse est vraie et nous mettons x dans D, autrement, nous mettons x dans G. Là encore, les invariants sont vérifiés.

A la fin de cette étape, on peut dire que les variables dans I n'ont pas de contraintes entre elles. Cependant, elles peuvent être impliquées dans des contraintes avec des variables de G ou de D. Dans la deuxième étape, nous passons en revue les variables $x \in I$, et nous générons une requête où toutes les variables de G sont mises à 0, toutes les variables de D sont mises à 1 et x est mise à 0. Si la réponse est *non*, on peut dire qu'il existe une contrainte $y < x$ avec $y \in G$ et donc x est ajoutée à D (et retirée de I). Sinon, nous posons la même requête, mais avec la valeur de x qui passe à 1. Si la réponse est *non*, il existe alors un $y \in D$ tel que $x < y$ appartient au réseau cible, donc x est ajoutée à D (et retirée de I). Pour le dernier cas où la réponse est *oui* pour les deux requêtes, on peut conclure que x n'appartient à aucune contrainte du réseau cible. Lorsque chaque variable a été examinée de cette manière, les variables restantes dans I ne sont pas impliquées dans les contraintes du réseau cible.

□

Theorem 4. QuAcq *n'apprend pas les réseaux de contraintes booléennes dans le langage* $\{<\}$ *avec un nombre minimal de requêtes.*

Démonstration. D'après le théorème 3, ces réseaux peuvent être appris en $O(n)$ requêtes. Ce type de réseaux peut contenir jusqu'à $n-1$ contraintes non redondantes. QuAcq apprend une contrainte à la fois, où chaque appel à `FindScope` prend $\Omega(\log n)$ requêtes. Par conséquent, QuAcq a besoin de $\Omega(n \log n)$ requêtes. □

5 Evaluation Expérimentale

Pour évaluer la faisabilité de notre approche, nous avons produit des résultats expérimentaux avec QuAcq sur une machine Xeon E5462@2.80GHz Intel avec 16 Go de RAM, et sur plusieurs problèmes.

Random. Nous avons généré des réseaux de contraintes binaires de façon aléatoire, avec 50 variables, des domaines de taille 10, et m contraintes binaires. Les contraintes binaires sont générées à partir du langage $\Gamma = \{\geq, \leq, <, >, \neq, =\}$. QuAcq prend en entrée un biais B contenant le graphe complet de 7350 contraintes binaires de Γ. Les résultats présentent la moyenne de 100 exécutions de QuAcq avec deux densités différentes : $m = 12$ (sous-contraint) et $m = 122$ (transition de phase).

Règles de Golomb. (prob006 in [8]) Le réseau cible est formulé avec m variables correspondant aux m marques de la règle, et des contraintes d'arité variable. Nous présentons les résultats sur des règles de taille 8. QuAcq est initialisé avec un biais de 770 contraintes basé dans le langage $\Gamma = \{|x_i - x_j| \neq |x_k - x_l|, |x_i - x_j| = |x_k - x_l|, x_i < x_j, x_i \geq x_j\}$, comprenant des contraintes binaires, ternaires [3] et quaternaires.

Problème du Zèbre. Le problème du Zèbre de Lewis Carroll a une solution unique. Le réseau cible est formulé avec 25 variables, des domaines de 5 valeurs, 5 cliques de contraintes $\neq$ et 14 contraintes supplémentaires figurant dans la description du problème. Pour tester QuAcq sur ce problème, nous l'avons initialisé avec un biais B de 4450 contraintes unaires et binaires d'un langage Γ de 24 contraintes (contraintes arithmétiques et des contraintes de distance).

Sudoku. Le réseau cible du problème de Sudoku est formulé avec 81 variables, des domaines de taille 9 et 810 contraintes $\neq$ sur les lignes, les colonnes et les carrés. Nous avons initialisé QuAcq avec un biais B de 6480 contraintes binaires du langage $\Gamma = \{=, \neq\}$.

Pour chaque problème, nous rapportons la taille $|C_L|$ du réseau appris (qui peut être plus petit que le réseau cible en raison des contraintes redondantes), le nombre total de requêtes $\#q$, le nombre de requêtes d'appartenance (complètes) $\#q_c$ (c.-à-d. les requêtes de la ligne 6 de QuAcq), la taille moyenne des requêtes $\bar{q}$, et le temps moyen nécessaire pour générer une requête (en secondes).

5.1 QuAcq et convergence

Pour assurer une convergence rapide, nous avons besoin de requêtes classées positives et qui réduisent au maximum le biais B. Le meilleur moyen d'avoir de telles requêtes est de générer un exemple à la ligne 4 de QuAcq qui maximise les contraintes violées dans B. Nous avons implémenté l'heuristique *max* pour générer une solution du C_L courant qui viole un nombre maximum de contraintes de B. Toutefois, cette heuristique peut être couteuse puisqu'elle résout un problème d'optimisation. Nous avons ensuite ajouté des bornes de 1 et 10 secondes pour l'heuristique *max*, pour obtenir respectivement deux variantes *max-1* et *max-10*. Nous avons également implémenté une heuristique moins couteuse que nous appelons *sol*. Elle résout simplement le C_L courant et s'arrête à la première solution qui viole au moins une contrainte de B.

Notre première expérimentation était de comparer *max-1* et *max-10* sur de gros problèmes. On observe que les performances lorsqu'on utilise *max-1* n'est pas significativement meilleure en nombre de requêtes que *max-10*. Par exemple, sur rand_122, on a $\#q = 1074$ pour *max-1* et $\#q = 1005$ pour *max-10*. Le temps moyen pour générer une requête étant 0.14 secondes pour *max-1* et 0,86 pour *max-10* avec, respectivement,

3. Les contraintes ternaires sont obtenues lorsque $i = k$ ou $j = l$ dans $|x_i - x_j| \neq |x_k - x_l|$

TABLE 1 – Résultats de QUACQ (convergence).

		$\|C_L\|$	$\#q$	$\#q_c$	$\bar{q}$	temps
rand_12	*max-1*	12	196	34	24.04	0.23
	sol	12	286	133	33.22	0.09
rand_122	*max-1*	86	1074	94	13.90	0.14
	sol	83	1062	120	15.64	0.06
Golomb	*max-1*	91	488	101	5.12	0.32
	sol	138	709	153	5.31	0.25
Zèbre	*max-1*	60	638	64	8.22	0.15
	sol	60	634	63	8.20	0.02
Sudoku	*max-1*	810	8645	821	20.58	0.16
	sol	810	9593	815	20.84	0.06

une borne de 1 et 10 secondes. Nous avons donc décidé de ne pas présenter les résultats avec *max-10*.

Le tableau 1 présente les résultats obtenus avec QUACQ pour apprendre un réseau de contraintes jusqu'à convergence en utilisant les heuristiques *max-1* et *sol* . Une première observation est que l'heuristique *max-1* nécessite généralement moins de requêtes que *sol* pour atteindre la convergence. Cela est particulièrement vrai pour rand_12, qui est très creux, et Golomb, qui contient de nombreuses contraintes redondantes. Si on regarde de plus près, cette différence s'explique principalement par le fait que *max-1* nécessite beaucoup moins de requêtes complètes positives que *sol* pour réduire B à vide et prouver la convergence (rand_12 : 22 requêtes complètes positives pour *max-1* et 121 pour *sol*). Mais en général, *sol* n'est pas aussi mauvais que nous pouvions espérer. La raison en est que, exceptés les réseaux très creux, le nombre de contraintes de B violées «par chance» avec *sol* est assez grand. La deuxième observation est que lorsque le réseau contient un grand nombre de contraintes redondantes, *max-1* converge sur un réseau plus petit que *sol* . Nous avons observé cela sur Golomb, et d'autres problèmes non rapportés ici. La troisième observation est que la taille moyenne des requêtes est toujours significativement plus petite que le nombre de variables du problème. Une dernière observation est que *sol* est très rapide en termes de génération de requêtes. Il est donc envisageable d'utiliser QUACQ sur des problèmes réels de taille importante.

Notre deuxième expérimentation consiste à évaluer l'effet de la taille du biais sur le nombre de requêtes. Sur le problème du zèbre, nous avons initialisé QUACQ avec des biais de différentes tailles et stocké le nombre de requêtes pour chaque exécution. La figure 1 montre que lorsque $|B|$ augmente, le nombre de requêtes suit une échelle logarithmique. Un résultat prometteur car cela signifie que l'acquisition de contraintes avec des biais expressifs (de grande taille) passe à l'échelle.

QUACQ a deux principaux avantages comparant à CONACQ. Le premier est la taille moyenne des requêtes $\bar{q}$ avec les requêtes partielles, qui sont probablement plus facile à classifier pour l'utilisateur. Le second avantage est le temps que nécessite QUACQ pour générer des requêtes. CONACQ.2 a besoin de trouver des exemples qui violent exactement une contrainte du biais pour atteindre la convergence (ce qui peut être coûteux à calculer). D'autre part, QUACQ peut utiliser des heuristiques pas très couteuses comme *max-1* et *sol* pour générer des requêtes.

5.2 QuAcq as a solver

CONACQ.2 et MODELSEEKER ont besoin d'exemples positifs pour apprendre un réseau de contraintes. Par contre, QUACQ peut apprendre sans exemple positif complet. Cette propriété peut être très utile lorsque le problème n'a pas été préalablement résolu. Nous avons simplement besoin de sortir de QUACQ dès qu'un exemple complet est classé par l'utilisateur comme positif. Nous avons évalué cette fonctionnalité en résolvant une séquence de 5 instances de Sudoku, des grilles avec des cases pré-remplies. Pour chaque grille, on sort de QUACQ lorsque la solution est trouvée. Comme le but n'est pas d'atteindre la convergence, nous avons remplacé l'heuristique *max-1* par un *min-1*, une heuristique qui tente de satisfaire autant que possible les contraintes de B, avec une borne d'une seconde. Chaque exécution prend en entrée le C_L et le B résultants de l'exécution précédente, vu que le réseau partiellement appris d'une exécution donnée reste valide comme point de départ d'une résolution d'une autre grille. Le nombre de requêtes nécessaires pour résoudre chacune des 5 grilles est, respectivement, 3859, 1521, 687, 135 et 34. La taille de C_L après chaque exécution est, respectivement, 340, 482, 547, 558, et 561. Nous notons, que pour la première grille, où QUACQ commence avec un biais complet, nous trouvons la solution en seulement 44% de requêtes nécessaires pour que QUACQ converge (voir le tableau 1). Après cela, chaque exécution de QUACQ sur les grilles qui restent, a besoin de moins de requêtes pour trouver une solution vu que C_L se rapproche de plus en plus du réseau cible.

6 Conclusion

Nous avons proposé QUACQ, un algorithme qui apprend un réseau de contraintes en demandant à l'utilisateur de classifier des affectations partielles en tant que positive ou négative. Chaque fois que l'algorithme reçoit un exemple négatif, l'algorithme dé-

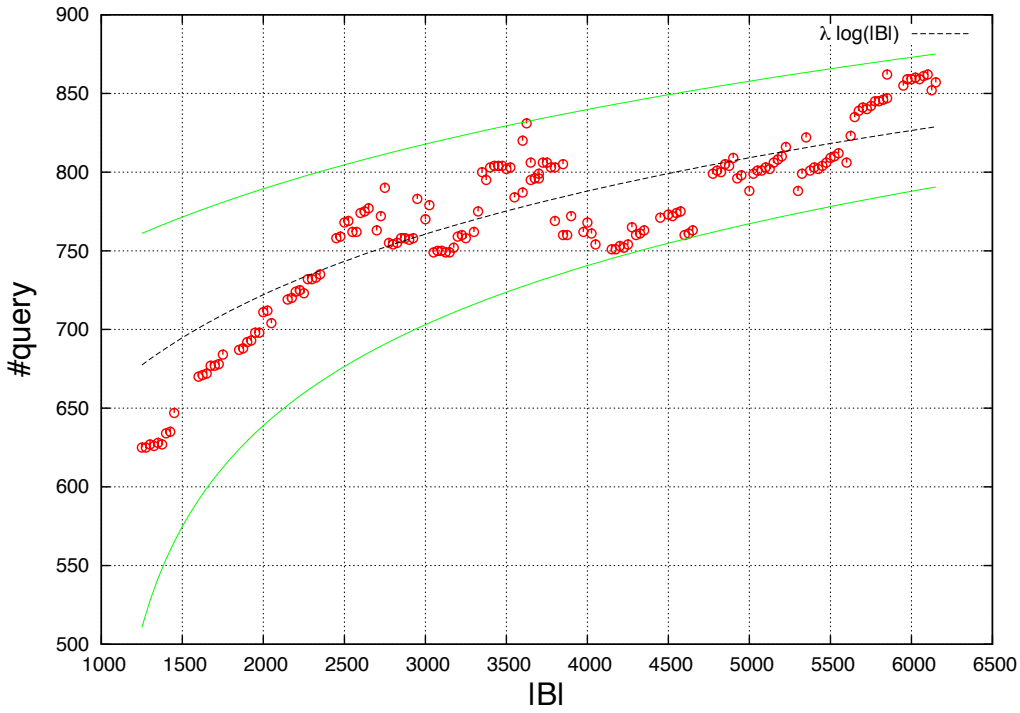


FIGURE 1 – QUACQ avec différentes tailles de biais sur le problème du zèbre

couvre une contrainte au problème en effectuant un nombre logarithmique de requêtes. Nous avons démontré que QUACQ est optimal pour certains langages de contraintes. Notre approche comporte quelques avantages par rapport aux travaux existants. Premièrement, elle converge toujours sur le réseau de contraintes cible en un nombre polynomial de requêtes. Deuxièmement, les requêtes sont souvent plus courtes que les requêtes d'appartenance et sont plus faciles à répondre par un utilisateur. Troisièmement, contrairement aux autres techniques, l'utilisateur n'a pas à fournir des exemples positifs pour converger. Cette dernière propriété peut être très utile lorsque le problème n'a pas été préalablement résolu. Nos expérimentations démontrent que générer de bonnes requêtes par QUACQ n'est pas difficile d'un point de vue computationnel et que lorsque le biais croît, le nombre de requêtes augmente de façon logarithmique. Ces résultats sont prometteurs pour l'utilisation de QUACQ sur des problèmes réels. Cepencdant, les problèmes avec des réseaux de contraintes denses (tel le Sudoku) requièrent un trop grand nombre de requêtes pour être répondu par un humain. Il serait intéressant d'utiliser MODELSEEKER pour apprendre rapidement des contraintes globales et d'utiliser QUACQ pour finaliser le modèle.

Références

[1] D. Angluin. Queries and concept learning. *Machine Learning*, 2(4) :319–342, 1987.

[2] N. Beldiceanu and H. Simonis. A model seeker : Extracting global constraint models from positive examples. In *Proceedings of the Seventeenth International Conference on Principles and Practice of Constraint Programming (CP'12), LNCS 7514, Springer–Verlag*, pages 141–157, Quebec City, Canada, 2012.

[3] C. Bessiere, R. Coletta, E. Freuder, and B. O'Sullivan. Leveraging the learning power of examples in automated constraint acquisition. In *Proceedings of the Tenth International Conference on Principles and Practice of Constraint Programming (CP'04), LNCS 3258, Springer–Verlag*, pages 123–137, Toronto, Canada, 2004.

[4] C. Bessiere, R. Coletta, F. Koriche, and B. O'Sullivan. A SAT-based version space algorithm for acquiring constraint satisfaction problems. In *Proceedings of the European Conference on Machine Learning (ECML'05), LNAI 3720, Springer–Verlag*, pages 23–34, Porto, Portugal, 2005.

[5] C. Bessiere, R. Coletta, B O'Sullivan, and M. Paulin. Query-driven constraint acquisition. In *Proceedings of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI'07)*, pages 44–49, Hyderabad, India, 2007.

[6] N.G. De Bruijn. *Asymptotic Methods in Analysis*. Dover Books on Mathematics. Dover Publications, 1970.

[7] E.C. Freuder and R.J. Wallace. Suggestion strategies for constraint-based matchmaker agents. In *Proceedings of the Fourth International Conference on Principles and Practice of Constraint Programming (CP'98), LNCS 1520, Springer–Verlag*, pages 192–204, Pisa, Italy, 1998.

[8] I.P. Gent and T. Walsh. Csplib : a benchmark library for constraints. http ://www.csplib.org/, 1999.

[9] U. Junker. Quickxplain : Preferred explanations and relaxations for over-constrained problems. In *Proceedings of the Nineteenth National Conference on Artificial Intelligence (AAAI'04)*, pages 167–172, San Jose CA, 2004.

[10] A. Lallouet, M. Lopez, L. Martin, and C. Vrain. On learning constraint problems. In *Proceedings of the 22nd IEEE International Conference on Tools for Artificial Intelligence (IEEE-ICTAI'10)*, pages 45–52, Arras, France, 2010.

[11] M. Paulin, C. Bessiere, and J. Sallantin. Automatic design of robot behaviors through constraint network acquisition. In *Proceedings of the 20th IEEE International Conference on Tools for Artificial Intelligence (IEEE-ICTAI'08)*, pages 275–282, Dayton OH, 2008.

Traduction de jeux à information incertaine en réseaux de contraintes stochastiques

Frédéric Koriche, Sylvain Lagrue, Éric Piette, Sébastien Tabary
Université Lille-Nord de France CRIL - CNRS UMR 8188 Artois, F-62307 Lens
{koriche,lagrue,epiette,tabary}@cril.fr

Résumé

Cet article propose d'utiliser le paradigme de la programmation par contraintes stochastiques pour modéliser et identifier des politiques optimales dans les jeux à information incertaine. Nous présentons une traduction permettant de modéliser les jeux décrits dans le formalisme GDL (Game Description Language) en instances du problème d'optimisation de contraintes stochastiques (SCSP). Notre traduction est démontrée correcte pour la classe GDL des jeux à information complète et environnement « indifférent ». L'intérêt de notre approche est illustré par une première résolution d'un jeu GDL en utilisant un solveur SCSP générique.

1 Introduction

La capacité pour un programme de jouer efficacement à n'importe quel jeu de stratégie (GGP pour *General Game Playing*) constitue l'un des défis majeurs de l'IA. Une compétition sur ce sujet est d'ailleurs organisée annuellement par AAAI [8]. Cette capacité générique de jouer a amené les chercheurs à confronter diverses approches efficaces, parmi lesquelles on peut citer des programmes utilisant des méthodes de type Monte Carlo [7], la construction automatique de fonctions d'évaluation [5], ou encore la programmation logique [16] et ASP [13]. Bien entendu, le problème GGP n'est pas cantonné aux « univers ludiques » : il permet de modéliser des problèmes de prise de décision séquentielle mono ou multi-agents.

Dans le contexte GGP, les jeux sont décrits dans un langage de représentation, appelé GDL pour *Game Description Language* [12]. Ce langage, basé sur la programmation logique, permettait dans sa première version de modéliser n'importe quel jeu à information certaine et complète. Une nouvelle version du langage, GDLII (GDL *with Incomplete Information* [17]), permet quant à elle de traiter des jeux à informations incertaines et/ou incomplètes. Dans ce cadre, et de façon usuelle en théorie des jeux, l'incertitude est modélisée par l'intervention de l'environnement, qui est un joueur comme les autres. Les actions de ce joueur, dénommé `random`, ne sont guidées par aucun objectif, mais juste par une distribution de probabilités entre les actions possibles. Il est ainsi possible par ce biais de modéliser des lancers de dés ou de pièces de monnaie, ou encore des distributions de cartes.

Différents formalismes ont été proposés afin de gérer les problèmes liés aux jeux à l'aide de réseaux de contraintes parmi lesquels on peut citer les QCSP [9], les *strategic constraint satisfaction problems* [3] ou encore les *constraint games* [14]. Mais aucun d'entre eux n'aborde les problèmes liés à l'aléatoire ni ne fait le lien avec GDL. L'objectif de cet article est d'aborder le problème GGP d'un point de vue original, fondé sur les réseaux de contraintes stochastiques (SCSP) [18]. En effet, les SCSP sont des outils puissants permettant non seulement de modéliser et de résoudre des jeux à infor-

mations complètes et certaines via QCSP [2], mais également de modéliser l'intervention du hasard. Nous montrons comment réécrire des jeux décrits à l'aide de GDL en SCSP et, par ce biais, comment la résolution du problème correspondant peut identifier une stratégie qui, en espérance, est optimale.

L'article est organisé de la façon suivante. Le formalisme GDL est introduit en Section 2. Nous présentons en particulier une sémantique originale, basée sur les *jeux stochastiques* (ou *jeux de Markov*) [15]. Nous présentons également dans cette section un jeu à information incertaine, le jeu du Verger, qui sert de fil conducteur pour l'ensemble de cet article. Après avoir rappelé en Section 3 le formalisme SCSP, la section 4 propose une traduction (prouvée correcte) d'un fragment de GDL vers SCSP. Avant de conclure, des expérimentations préliminaires utilisant un solveur générique SCSP sont présentées en Section 5.

2 Formalisme GDL

L'objectif de GDL est de fournir un langage générique pour la représentation de n'importe quel jeu, y compris les jeux collaboratifs ou encore les jeux à actions simultanées. La version récente [17] permet de traiter des jeux avec observation partielle où le hasard peut intervenir. Notre étude se focalise sur un fragment particulier de cette version : les jeux à information complète et environnement « indifférent » (*oblivious environment*). Pour ces jeux, les joueurs ont une observation complète de l'état courant. L'environnement est aléatoire mais son comportement ne peut pas être influencé par celui des (autres) joueurs. Ce fragment de GDLII couvre une classe de problèmes très étudiée en théorie des jeux [4], ainsi qu'une variété de jeux de stratégie pour lesquels les joueurs n'ont pas d'effet sur les actions de l'environnement. Un exemple caractéristique est celui des « jeux de dés » où les joueurs peuvent choisir les dés qu'ils lancent, mais ne peuvent pas influencer le comportement des dés.

2.1 Langage

Le langage GDL est dérivé de la programmation logique (avec égalité et négation). Rappelons que l'univers de Herbrand d'un programme logique est défini par l'ensemble des termes instanciés (*grounded terms*) obtenus à partir des symboles de fonctions (incluant les constantes) du programme. Dans un programme GDL, les *joueurs* et *objets* du jeu sont décrits par des constantes, tandis que les *fluents* (ou percepts) et *actions* (coups) sont décrits par des termes. Par exemple, dans le jeu du morpion, le terme $\texttt{cell}(2,2,\texttt{b})$ est un fluent indiquant que la case $(2,2)$ du plateau est marquée par un jeton noir. Nous notons Σ l'univers de Herbrand d'un programme GDL ; Σ_F et Σ_A désignent les sous-ensembles de Σ correspondant respectivement aux termes des fluents et aux termes des actions.

Les atomes d'un programme GDL sont construits à partir d'un ensemble fini de symboles de relations et de symboles de variables. Certains symboles ont un rôle spécifique dans le programme, et sont décrits dans la table 1.

Mot-clé	Description
role(J)	J est un joueur
init(F)	le fluent F fait partie de l'état initial
true(F)	F fait partie de l'état courant
legal(J,A)	J peut faire l'action A
does(J,A)	l'action de J est A
next(F)	F fait partie de l'état suivant
terminal	l'état courant est terminal
goal(J,N)	J reçoit N dans l'état courant
random	le « joueur » environnement

Table 1 – Mots-clés GDL

Par exemple, $\texttt{legal}(\texttt{P},\texttt{mark}(\texttt{X},\texttt{Y}))$ est un atome indiquant que le joueur P a le droit de marquer la case $(\texttt{X},\texttt{Y})$ du plateau.

Les règles d'un programme GDL sont des implications composées d'un atome pour le conséquent et d'un ensemble de littéraux (atomes ou négations d'atomes) pour l'antécédent. Par exemple, la règle :

$$\texttt{legal}(\texttt{random},\texttt{noop}) \leftarrow \texttt{true}(\texttt{control}(\texttt{bob}))$$

signifie que si le joueur bob a actuellement la main alors noop (ne rien faire) est une action légale du joueur random.

Afin d'être *valide*, un programme GDL doit satisfaire plusieurs conditions syntaxiques. Pour des raisons de décidabilité, il doit être *stratifié* [1] et *permis* [11] ; ces deux conditions ne sont pas détaillées pour des raisons de place, mais sont expliquées dans [12]. D'autre part,

comme le but d'un programme GDL est de représenter un modèle formel de jeu, il doit aussi respecter des conditions sur les mots-clés :

(i) `role` apparaît uniquement dans les faits ;

(ii) `true` apparaît uniquement dans le corps des règles ;

(iii) `init` apparaît uniquement dans la tête des règles et ne peut dépendre des mots-clés `true`, `legal`, `does`, `next`, `terminal` et `goal` ;

(iv) `does` apparaît uniquement dans le corps des règles et ne dépend pas de `legal`, `terminal` et `goal` ;

(v) `next` apparaît uniquement dans la tête des règles.

Dans le cadre des jeux à information complète et environnement indifférent, nous ajoutons les conditions :

(vi) Pour chaque fluent `F`, il existe une instance `f` de `F` tel que `init(f)` est un fait.

(vii) Les atomes `legal(random, A)` où `A` désigne une action apparaissent uniquement dans les faits.

2.2 Sémantique

Un modèle courant pour représenter les jeux à information incertaine est celui des *jeux stochastiques* (ou *jeu de Markov*) [15]. Un jeu de Markov à n joueurs sur Σ est un tuple $\langle N, S, \boldsymbol{A}, P, \boldsymbol{R}\rangle$ tel que : [1]

— $N = \{1, \ldots, n\}$ est l'ensemble des *joueurs*,

— S est un ensemble d'*états* ;

— $\boldsymbol{A} = A_1 \times ... \times A_n$, où A_i est un ensemble fini d'actions accessibles au joueur i ;

— $P : S \times \boldsymbol{A} \times S \to [0, 1]$ est la fonction de transition ; $P(s, \boldsymbol{a}, s')$ est la probabilité de passer de l'état s en l'état s' en appliquant l'action jointe $\boldsymbol{a}$;

— $\boldsymbol{R} = \langle r_1, ..., r_n\rangle$, où $r_i : S \to \mathbb{R}$ est la fonction de gain du joueur i.

À ce tuple s'ajoutent l'état *initial* $s_0 \in S$ et l'ensemble des états *terminaux* $S_{ter} \subseteq S$.

Montrons maintenant comment construire un tel modèle à partir d'un programme valide GDL, noté P. Un état du jeu est un sous-ensemble de son univers de Herbrand Σ. Puisque les restrictions syntaxiques de GDL garantissent une dérivabilité finie des termes instanciés, tout état est une partie *finie* de Σ_F. Les n instances de `role(R)` définissent les n joueurs du modèle stochastique. Désignons par $n + 1$ l'environnement (`random`).

L'état initial est construit à partir des termes de `init(F)`.

Afin de capturer les coups légaux d'un joueur dans un état donné $s = \{f_1, \cdots, f_m\}$, notons $s_{\texttt{true}}$ l'ensemble des faits $\{\texttt{true}(f_1), ..., \texttt{true}(f_m)\}$. Les termes instanciés de `legal(J, A)` dérivables de $\texttt{P} \cup s_{\texttt{true}}$, définissent toutes les actions légales des joueurs `J` dans l'état s. Les états terminaux $\mathcal{S}_{ter}$ et le vecteur de fonctions de gain $\boldsymbol{R}$ sont construits de manière analogue, en utilisant les atomes `terminal` et `goal(R, N)`. Dans un jeu à environnement indifférent, l'ensemble des coups légaux de $n+1$, noté $L(n+1)$, est indépendant de l'état courant : il est construit à partir des termes de `legal(random, A)`.

Pour spécifier la fonction de transition, nous avons besoin de l'état courant ainsi que des actions réalisées simultanément dans cet état. Pour un vecteur $\boldsymbol{a} = \langle a_1, \cdots, a_n, a_{n+1}\rangle$ d'actions jouées par les n joueurs et l'environnement $(n+1)$, notons $\boldsymbol{a}_{\texttt{does}}$ l'ensemble des faits : $\{\texttt{does}(1, a_1), ..., \texttt{does}(n + 1, a_{n+1})\}$. L'état suivant est construit à partir des termes instanciés de `next(F)` dérivables de $\texttt{P} \cup s_{\texttt{true}} \cup \boldsymbol{a}_{\texttt{does}}$. De manière concise, cet état suivant est noté $Q(s, \boldsymbol{a})$. En particulier, si $\boldsymbol{a}$ est la concaténation $\boldsymbol{a'} \cdot a_{n+1}$ d'une action jointe $\boldsymbol{a'}$ des n joueurs et d'une action a_{n+1} de l'environnement, alors $Q(s, \boldsymbol{a'} \cdot a_{n+1})$ dénote l'état suivant obtenu à partir de s lorsque, simultanément, les joueurs ont accompli $\boldsymbol{a'}$ et l'environnement à joué a_{n+1}.

La distribution de probabilités sur les transitions est capturée par le comportement stochastique de l'environnement : elle est définie par la distribution uniforme sur tous les coups légaux que peut jouer $n + 1$ dans l'état courant. Notons que les états résultants ne sont pas forcément équiprobables. [2]

1. Pour un ensemble U, nous notons 2^U l'ensemble de toutes les parties *finies* de U.

2. Par exemple, un dé « pipé » avec une probabilité de 1/2 de tomber sur 6 peut être modélisé par dix actions légales de `random`, dont cinq donnent le même effet (tomber sur un 6).

Définition 1 *La sémantique d'un jeu valide* GDL P *à environnement indifférent peut être décrite par le jeu stochastique* $\langle N, S, \boldsymbol{A}, P, \boldsymbol{R}\rangle$ *tel que :*

— $N = \{i : \mathtt{P} \models \mathtt{role(i)}\ \textit{et}\ \mathtt{i} \neq \mathtt{random}\}$;

— $S = 2^{\Sigma_F}$;

— $A_i = \{a : \mathtt{P} \cup s_{\mathtt{true}} \models \mathtt{legal(i,a)}, s \in S\}$;

— $P(s, \boldsymbol{a}, s') = \frac{|\{a_{n+1} \in L(n+1) : s' = Q(s, \boldsymbol{a} \cdot a_{n+1})\}|}{|L(n+1)|}$

— $r_i(s) = \begin{cases} c & \textit{si}\ \mathtt{P} \cup s_{\mathtt{true}} \models \mathtt{goal(i,c)}, \\ 0 & \textit{sinon.} \end{cases}$

Naturellement, l'état initial et les états terminaux sont définis par :

$$s_0 = \{f \in \Sigma_F : G \models \mathtt{init}(f)\}$$
$$S_{ter} = \{s \in S : G \cup s_{\mathtt{true}} \models \mathtt{terminal}\}$$

2.3 Un exemple : le Jeu du Verger

Le jeu du *Verger* (*Obstgarten*) est un jeu de société coopératif de 1 à 4 joueurs contre l'environnement. Il se compose de 4 arbres contenant chacun 10 fruits et d'un corbeau composé de 9 pièces. A chaque tour, chaque joueur lance un dé composé de 6 faces :

— 4 faces correspondant chacune à un arbre : le joueur doit alors retirer un fruit de cet arbre ;

— une face corbeau : le joueur doit retirer une pièce au corbeau ;

— une face panier : le joueur doit retirer deux fruits de son choix.

L'objectif est de retirer l'ensemble des fruits de chaque arbre avant que le corbeau ne soit entièrement décomposé. Il est intéressant de noter que la seule et unique décision du joueur intervient lors du tirage du « panier ». La stratégie optimale dans ce cas étant de toujours prendre les fruits dans l'arbre le plus chargé (ou les arbres les plus chargés). Des simulations montrent que cette stratégie permet de gagner dans environ 68,4% des cas, tandis que la pire stratégie (toujours prendre dans l'arbre le moins chargé) ne permet de gagner que dans 53,2% des cas.

Dans le but d'utiliser un exemple court, la figure 1 présente un codage GDL du *Verger* avec un seul joueur (bob), 2 arbres contenant 2 fruits et un corbeau de taille 1. Nous utiliserons pour ce jeu un dé à quatre faces : r (prendre un fruit dans l'arbre rouge), v (prendre un fruit dans l'arbre vert), p (laisser la main au joueur pour qu'il choisisse deux fruits) et c (retirer une pièce au corbeau).

```
% les rôles
role(bob)
role(random)

% opérations sur les entiers
succ(0,0)
succ(0,1)
succ(1,2)

% couleurs des arbres
tree(r)
tree(v)

% initialisation de l'état du jeu
init(state(2,2,1))
init(control(random))
legal(random,roll(D))

% définition des coups légaux du joueur
legal(bob,noop) ← true(control(random))
legal(bob,choice(C1,C2)) ← tree(C1),
                    tree(C2), true(control(bob))

% évolution du jeu
next(control(bob)) ← does(random,roll(p)),
                          true(control(random))
next(control(random)) ← true(control(bob))
next(control(random)) ← does(random,roll(R)),
                    true(control(random)), R ≠ p

next(state(R,V,C)) ← true(state(R1,V,C)),
                 succ(R,R1),does(random,roll(r))
...
next(state(R,V,C)) ← true(state(R1,V1,C)),
  succ(R,R1), succ(V,V1), does(bob,choice(v,r))

% objectifs et récompenses
goal(bob,100) ← true(state(0,0,C))

% fin du jeu
terminal ← true(state(R,V,0))
terminal ← true(state(0,0,C))
```

FIGURE 1 – Programme GDL du Verger allégé

3 CSP Stochastiques

Le modèle des CSP stochastiques que nous présentons dans cette étude, étend le cadre original de Walsh [18] aux contraintes valuées. De manière formelle, un *Stochastic Constraint Satisfaction Problem* (SCSP) est un 6-tuple $\langle X, Y, D, P, \mathcal{C}, \theta\rangle$ où X représente un ensemble de n variables, Y est un sous-ensemble de X représentant les variables stochastiques, D représente les domaines associés aux variables de X, P est l'ensemble des distributions de probabilités appliquées aux domaines des variables

stochastiques, $\mathcal{C}$ est l'ensemble des contraintes et θ représente une valeur de seuil dans $\mathbb{R}$.

Un SCSP est donc composé de variables de décisions et de variables stochastiques. Les variables de décisions ont le même sens que celles définies dans le cadre des CSP classiques. Pour chaque variable stochastique, on associe une distribution de probabilité aux valeurs des domaines de ces variables. Pour une variable x, on note $D(x)$ le domaine associé à x, et plus généralement, pour un sous-ensemble $Z = \{z_1, \cdots, z_k\}$ de variables dans X, on note $D(Z)$ l'ensemble des tuples de valeurs $D(z_1) \times \cdots \times D(z_k)$.

Chaque contrainte du SCSP est une paire $C = \langle scp_C, val_C \rangle$, où scp_C est un sous-ensemble de X, appelée *portée* de C, et val_C est une fonction qui associe à chaque tuple $\boldsymbol{\tau} \in D(scp_C)$ une valeur (ou utilité) $val_C(\boldsymbol{\tau})$ dans $\mathbb{R} \cup \{-\infty\}$. Une contrainte C est *dure* si son ensemble d'arrivée est $\{-\infty, 0\}$. Dans ce cas, C peut être représentée de manière habituelle par une relation, notée rel_C indiquant les tuples de valeurs autorisées (i.e. les tuples $\boldsymbol{\tau}$ pour lesquels $val_C(\boldsymbol{\tau}) \neq -\infty$). Dans un SCSP les contraintes dures portent uniquement sur des variables de décision. Les autres contraintes, dites *souples*, peuvent porter sur des variables arbitraires.

Une *interprétation* I est un tuple dans $D(X)$ qui associe donc à chaque variable $x \in X$ une valeur dans son domaine $D(x)$. Si l'on note $I_{|Z}$ la projection de I sur le sous-ensemble $Z \subseteq X$, alors l'utilité d'une interprétation I est :

$$val(I) = \sum_{C \in \mathcal{C}} val_C(I_{|scp_C})$$

Une *politique* π est représentée par un arbre dont chaque nœud décrit une variable de l'ensemble X. Les nœuds représentant des variables de décisions ne comportent qu'un seul fils (correspondant à la valeur assignée à cette variable) tandis que les nœuds représentant des variables stochastiques possèdent un fils pour chaque valeur possible du domaine. Les arêtes sont étiquetées par la valeur assignée à la variable correspondante. Les feuilles sont étiquetées par l'utilité de l'assignation associée au chemin depuis la feuille jusqu'à la racine. L'utilité espérée d'une politique π se définie alors comme la somme des utilités des feuilles pondérées par leurs probabilités. Une *solution* du SCSP est une politique π dont l'utilité espérée est supérieure ou égale au seuil θ.

Il est facile de voir que si une politique π contient au moins une branche dont l'assignation viole une contrainte dure, son utilité espérée ($-\infty$) ne peut pas satisfaire le SCSP. Ainsi une politique π satisfait le problème de contraintes stochastiques si elle est « faisable » (i.e. ne viole aucune contrainte dure) et son utilité espérée dépasse le seuil θ.

Considérons par exemple le SCSP de la figure 2. Dans la politique π qui est illustrée, les variables x et z sont décisionnelles (assignées à la valeur 0). La variable y est stochastique, avec trois valeurs (0, 1 et 2) équiprobables. Comme π satisfait systématiquement la contrainte dure C_h, et satisfait la contrainte souple C_s avec une utilité espérée de $2/3 \geq 1/2$, cette politique constitue donc une solution du SCSP.

$X = \{x, y, z\}$ $Y = \{y\}$ $\mathcal{C} = \{C_h, C_s\}$ $\theta = 1/2$	$D(x) = D(y) = D(z) = \{0, 1, 2\}$ $P(0) = P(1) = P(2) = 1/3$ $C_h : x = z$ $C_s(x, y) = \begin{cases} 1 & \text{si } x + y \geq 1 \\ 0 & \text{sinon} \end{cases}$

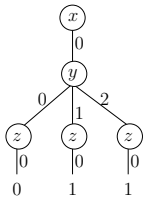


FIGURE 2 – Un SCSP et une politique π

4 De GDL à SCSP

Afin d'obtenir un modèle SCSP à partir d'un programme GDL, il est nécessaire de fixer au préalable un horizon de temps T, dont les pas $t \in \{1, \cdots, T\}$ capturent les tours de jeu. La traduction que nous proposons permet de construire à partir d'un programme GDL P et d'un horizon T, les ensembles de variables X, de domaines D et de contraintes $\mathcal{C}$, ainsi que les distributions de probabilité P du SCSP correspondant. Cette traduction se décompose en quatre étapes que nous allons examiner.

4.1 Élimination des fonctions

Les composants du SCSP associés à un programme GDL P sont extraits à partir des termes, atomes et règles de P. En particulier, les domaines et contraintes font appel aux termes instanciés dérivables de P. Même si avec les restrictions syntaxiques de GDL, il est possible d'utiliser l'approche de la Section 2 pour dériver les termes instanciés de P apparaissant comme fluents et actions, le coût de telles requêtes sur un programme logique stratifié est généralement prohibitif [6].

Afin de circonvenir à ce problème, notre traduction utilise une transformation de P en un programme P′ sans fonction. En utilisant la méthode de Lifschitz et Yang [10], cette transformation se déroule en deux phases : un aplatissement (*flattening*) qui réduit les imbrications de termes, suivi d'une élimination qui remplace les fonctions par des prédicats.

L'aplatissement procède comme suit : pour chaque symbole de fonction f apparaissant dans P, on construit un programme équivalent $\mathtt{P_f}$ dit f-*aplati* dans lequel toute occurrence d'un terme de la forme $\mathtt{f}(\mathtt{t_1}, \cdots, \mathtt{t_k})$ est transformée par une égalité de la forme $\mathtt{f}(\mathtt{t_1}, \cdots, \mathtt{t_k}) = \mathtt{Z}$, où Z est une variable de renommage. Par exemple, si P contient l'atome $\mathtt{legal}(\mathtt{random}, \mathtt{roll}(\mathtt{c}))$, alors dans $\mathtt{P_f}$ l'atome est transformé en la conjonction $\mathtt{legal}(\mathtt{random}, \mathtt{Z}), \mathtt{roll}(\mathtt{c}) = \mathtt{Z}$. En appliquant le f-aplatissement sur toutes les fonctions f de P, le programme obtenu ne contient plus d'imbrication. Notons que l'aplatissement est polynomialement borné par le nombre de fonctions et la profondeur des termes.

La phase d'élimination s'effectue de la manière suivante : chaque symbole de fonction f d'arité k est associé à un symbole de relation F d'arité $k+1$; toute égalité de la forme $\mathtt{f}(\mathtt{t_1}, \cdots, \mathtt{t_k}) = \mathtt{z}$ est remplacée par l'atome $\mathtt{F}(\mathtt{t_1}, \cdots, \mathtt{t_k}, \mathtt{z})$; enfin, la règle [3] :

$$(\exists!\mathtt{z})\mathtt{F}(\mathtt{t_1}, \cdots, \mathtt{t_k}, \mathtt{z})$$

est ajoutée à P′ garantissant ainsi l'équivalence des modèles stables entre P et P′.

3. En programmation logique ∃! signifie "il existe un et un seul".

4.2 Extraction des variables

L'ensemble X des variables du SCSP est obtenu de la manière suivante. On associe à $t \in \{1, \cdots, T\}$, les variables $role_t$, $control_t$, et $terminal_t$. Intuitivement, $role_t$ indique le joueur représenté au tour t, $control_t$ indique quel joueur à la main au tour t, et $terminal_t$ indique si au temps t le jeu est terminé.

Pour chaque instance i de role(J) et chaque $t \in \{1, \cdots, T-1\}$ est engendrée une variable $a_{i,t}$ indiquant l'action du joueur i au tour t. De même, une variable $a_{random,t}$ est construite pour l'action de l'environnement.

Les variables SCSP associées aux fluents sont extraites à partir des prédicats décrivant les fluents du programme P′. Spécifiquement, si $\mathtt{F}(\mathtt{X_1}, \cdots, \mathtt{X_k}, \mathtt{Z})$ est un atome d'arité $k+1$, tel que la variable de renommage apparaît dans un des prédicats init, true et next, alors F est un fluent et la variable f_t est ajoutée à X.

Enfin, comme un programme GDL peut inclure des symboles de relation dits « statiques », qui ne varient pas d'état en état, mais sont utilisés pour établir des relations entre fluents (e.g. succ(I,J)). Pour chaque symbole de relation statique et instant, une variable SCSP est construite. Cependant, comme ces variables sont universelles sur toutes les contraintes où elles apparaissent, on pourra alors dans une phase de prétraitement les retirer de la portée des contraintes.

4.3 Extraction des domaines

Concernant les variables associées au déroulement du jeu, $terminal_t$ est booléenne, et le domaine de $role_t$ et $control_t$ est l'ensemble constitué par les N joueurs (extraits préalablement) et l'environnement (random).

Le domaine de chaque variable fluent f est donné par l'ensemble des combinaisons des constantes $\mathtt{c_1}, \cdots, \mathtt{c_k}$ qui peuvent être instanciées des atomes A de la forme $\mathtt{F}(\mathtt{t_1}, \cdots, \mathtt{t_k}, \mathtt{z})$. Ces domaines sont extraits par filtrage sur le réseau suivant. Pour chaque fluent F d'arité $k+1$ nous construisons les variables $f_1^v, \cdots, f_k^v$ et $f_1^c, \cdots, f_k^c$. Les domaines de $f_1^v, \cdots, f_k^v$ sont initialement formés par tous les symboles de constantes du programme P et le domaine de chaque f_i^c est formé par l'ensemble des symboles de constantes apparaissant en position i

GDL	Variable SCSP	Domaine SCSP
role(J)	$role_t$	{random, bob, undefined}
legal(bob,A) legal(random,A)	$a_{bob,t}$ $a_{random,t}$	{{choice} × {r, v} × {r, v}} ∪ {noop, undefined}} {{roll} × {c, p, r, v}}
next(state(...)) next(control(...))	$state_t$ $control_t$	{(3, 3, 2), ..., (0, 0, 0)} ∪ {undefined} {bob, random, undefined}
succ(...) tree(...)	$succ_t$ $tree_t$	{(0, 0), (0, 1), (1, 2), (2, 3), undefined} {r, v, undefined}
terminal(...)	$terminal_t$	{true, false}

TABLE 2 – Variables et domaines au temps t associés au jeu du *Verger*

dans les atomes préfixés par F de P. Une arête (f_i^v, f_i^c) est construite pour chaque index i, et une arête (f_i^v, g_j^v) est construite dès lors que, dans une règle de P, le même symbole de variable apparaît en position i dans un atome préfixé par F et en position j par un atome préfixé par G. Par arc-consistance, on élimine dans chaque variable f_i^v les constantes ne possédant pas de support. Ainsi, les valeurs de f_i^v sont données par l'union des valeurs de f_i^c et celles des variables voisines g_j^v possédant un support. Enfin, pour chaque instant t, le domaine de la variable fluent f_t est donné par $D(f_t) = D(f_1^v) \times \cdots \times D(f_k^v)$.

L'extraction des domaines des variables d'action s'effectue de manière analogue, en identifiant au préalable, à partir de la relation legal(j,A), les atomes A qui participent à la construction du domaine de $a_{j,t}$.

Un point technique : on rajoute la valeur undefined à chaque domaine des variables de décisions car si $terminal_t$ prend la valeur true, toutes ces variables différentes de *terminal* à un instant supérieur à t sont instanciées à undefined.

La table 2 illustre l'extraction des variables et des domaines obtenues sur le jeu du *Verger* présenté dans la section 2.3.

4.4 Extraction des contraintes

Pour chaque règle R du programme GDL P et chaque instant t, on associe une contrainte $C_{R,t}$. Les règles de réécritures de la table 3 spécifient la portée des contraintes.

Dans le SCSP, toutes les contraintes sont dures, exceptée la contrainte de score. Ainsi, pour chaque règle R du programme P, la sémantique de $C_{R,t}$ est une « relation » sur le domaine de sa portée. Cette relation est construite comme suit. Après élimination de

Atome GDL	Portée de la contrainte
init(f(...))	$\{f_0\} \in scp_{C_0}$
true(f(...))	$\{f_t\} \in scp_{C_t}$
does(j,a(...))	$\{a_{j,t}\} \in scp_{C_t}$
next(f(...))	$\{f_{t+1}\} \in scp_{C_t}$
legal(j,a(...))	$\{a_{j,t}\} \in scp_{C_t}$
goal(j,N)	$\{role_t\} \in scp_{score_t}$
terminal	$\{terminal_t\} \in scp_{terminal}$

TABLE 3 – règles de réécritures de la portée des contraintes

fonctions, nous savons que tous les termes de R dans P ont été transformés en prédicats dans la règle correspondante R′ de P′. Supposons que cette règle résultante R′ soit de la forme :

$$\mathtt{A_1}, \cdots, \mathtt{A_k} \leftarrow \mathtt{B_1}, \cdots, \mathtt{B_{k'}}$$

Notons qu'après élimination des fonctions, l'antécédent peut contenir plusieurs atomes. Comme R′ exprime en général une implication, la relation de $C_{R,t}$ doit capturer cette implication. Dans ce but, soit A (resp. B) l'ensemble $\{\mathtt{A_1}, \cdots, \mathtt{A_k}\}$ (resp. $\{\mathtt{B_1}, \cdots, \mathtt{B_{k'}}\}$). Soit U($\mathtt{A_i}$) l'ensemble des combinaisons de constantes qui sont instances de $\mathtt{A_i}$, et U(A) (resp. U(B)) la jointure de ces combinaisons sur A (resp. B). D'autre part, soient C(B) (resp. C(AB)) l'ensemble des combinaisons de constantes qui sont instances de la conjonction sur B (resp. A ∪ B). L'ensemble des combinaisons de constantes qui sont instances de R′ est donc :

$$\mathtt{C(AB)} \cup (\mathtt{U(A)} \bowtie (\mathtt{U(B)} \setminus \mathtt{C(B)}))$$

Cet ensemble d'instances peut être obtenu par application de requêtes conjonctives. La projection de cet ensemble sur $scp_{C_{R,t}}$ donne $rel_{C_{R,t}}$.

Notons que par la règle d'information complète (vi), les contraintes associées à init n'autorisent qu'une seule valeur par variable de fluent f_0. La contrainte souple $score_t$ associe à chaque joueur j la valeur N donnée dans

goal(j,N) si le corps de la règle associée à cet atome est vrai, et la valeur 0 sinon.

Pour terminer, la composante P du SCSP est formée en associant une distribution de probabilité uniforme sur le domaine de $random_t$ à chaque instant t.

4.5 Équivalence des modèles

Dans le modèle SCSP obtenu, notons $X_{f,0}$ l'ensemble des variables de fluent apparaissant dans l'état initial, et $X_{f,t}$ (resp. $X_{a,t}$) l'ensemble des variables de fluent (resp. d'action) au temps t. Par construction,

$$X = \left(\bigcup_{t=0}^{T} X_{f,t}\right) \cup \left(\bigcup_{t=1}^{T-1} X_{a,t}\right)$$

La réécriture obtenue détermine un graphe orienté sans circuit (DAG) étiqueté $G_{\text{SCSP},T}$ dont l'ensemble des sommets S et des arcs A est le plus petit ensemble construit de la manière suivante :

— l'unique tuple $s_0 \in D(X_{f,0})$ qui est autorisé par le SCSP est un sommet de S ;

— si $s_t \in D(X_{f,t})$ est un sommet de S et $s_{t+1} \in D(X_{f,t+1})$ tel que la concaténation $s_t \cdot s_{t+1}$ est un tuple autorisé par le SCSP, alors s_{t+1} est un sommet de S et (s_t, s_{t+1}) est un arc de A.

Chaque arc $(\boldsymbol{s}_t, \boldsymbol{s}_{t+1})$ est étiqueté par la probabilité $\frac{k}{d}$ où d est la taille de $D(a_{\texttt{random},t})$ et k est le nombre de tuples $(\boldsymbol{s}_t, \texttt{a}, \boldsymbol{s}_{t+1})$ autorisés pour lesquels $\texttt{a} \in D(a_{\texttt{random},t})$. Enfin, chaque sommet s_t est étiqueté par une table associant à chaque joueur i dans le domaine de $role_t$ son utilité donnée par $score_t(i)$.

De manière analogue, nous associons à un programme GDL, noté P, le DAG suivant : chaque état s du jeu de Markov de P est réécrit en éliminant les symboles de fonctions (cf Section 4.1) et en ne conservant que la combinaison de constantes du prédicat ; l'ensemble des sommets est l'ensemble S des états (transformés) du jeu de Markov de $\texttt{P}'$ atteignables depuis l'état initial par un chemin de longueur au plus $T-1$.[4] Il existe un arc (s_t, s_t+1) entre deux sommets ssi il existe un vecteur d'actions légales $\boldsymbol{a}$ pour s_t, tel que $s_{t+1} = Q(s_t, \boldsymbol{a})$. Chaque arc (s_t, s_t+1) est étiqueté par la proportion d'actions a de **random** pour lesquelles $s_{t+1} = Q(s_t, \boldsymbol{a}\cdot\texttt{a})$. Enfin, chaque sommet s_t est étiqueté par son vecteur de récompenses $\boldsymbol{R}(s_t)$.

Proposition 1 *Pour tout horizon T, si SCSP est la traduction du programme GDL* P *alors les graphes $G_{SCSP,T}$ et $G_{\texttt{P},T}$ sont identiques.*

Preuve (schéma général). De part la règle de réécriture associée à init, le sommet s_0 de $G_{\text{SCSP},T}$ appartient aussi à $G_{\texttt{P},T}$. Par hypothèse d'induction, supposons que s_t appartienne à la fois à $G_{\text{SCSP},T}$ et $G_{\texttt{P},T}$. Considérons un état $s_{t+1} \in D(X_{f,t+1})$ tel que $s_t \cdot s_{t+1}$ est un tuple autorisé par le SCSP. Alors il existe un tuple $\boldsymbol{a} \in D(X_{a,t})$ tel que $s_t \cdot \boldsymbol{a}\boldsymbol{s}_{t+1}$ est autorisé par le SCSP, ce qui implique que $\boldsymbol{a}$ est un vecteur d'actions légales, et donc s_{t+1} appartient à la fois à $G_{\text{SCSP},T}$ et $G_{\texttt{P},T}$. De la même manière, l'arc (s_t, s_{t+1}) appartient aux deux graphes. Comme l'environnement est indifférent et que son domaine d'actions est le même pour P et SCSP, les étiquettes sur les arcs sont identiques. Enfin, par la réécriture de **goal**(j,N) en $score_t$, les étiquettes sur les sommets sont identiques.

5 Résolution et expérimentations préliminaires

Pour simplifier la résolution du réseau de contraintes stochastiques obtenu, nous utilisons des techniques de prétraitement.

La première technique utilisée est la fusion des contraintes de même portée. Ainsi, deux contraintes c_i et c_j (de relation rel_{c_i} et rel_{r_j}) tel que $scp_{c_i} = scp_{c_j}$ deviennent une unique contrainte c_k de relation $rel_{r_k} = rel_{c_i} \cup rel_{c_j}$.

La seconde technique utilisée est la suppression de toutes les contraintes unaires (d'arité 1). Le domaine des variables présentes dans ces contraintes est restreint aux valeurs satisfaisant les tuples de la relation associée.

Une dernière technique consiste en la détection des variables universelles dans chaque contrainte. Quelque soit la valeur prise par ces variables, la contrainte est toujours satisfaite. Ces variables sont alors supprimées de la portée des contraintes. Une variable x est universelle dans une contrainte c_i si le nombre de

4. La longueur du chemin est donnée par le nombre de ses arcs.

tuples de rel_{c_i} est égal au produit de la taille du domaine de x avec le nombre de tuples de la relation associée à la contrainte c_j telle que $scp_{c_i} \setminus \{x\} = scp_{c_j}$.

Algorithme 1 : search

```
Données : seuil θ_h politique π, entier i
Résultat : politique π, satisfaction θ
1  si i > |X| alors
2      retourner (π, 1)
3  θ ← 0
4  pour chaque v ∈ D(x_i) faire
5      si isConsistent(x_i, v, C) alors
6          si x_i ∈ S alors
7              p ← prob(x_i, v)
8              (π, θ_t) ← search((θ_h − θ)/p, π, i + 1)
9              θ ← θ + p ∗ θ_t
10             si θ ≥ θ_h alors
11                 retourner (π, θ)
12         sinon
13             π ← sol ∪ {(x_i, v)}
14             (π, θ_t) ← search(θ_h, π, i + 1)
15             θ ← max(θ, θ_t)
16             si θ ≥ θ_h alors
17                 retourner (π, θ)
18             sinon
19                 π ← π \ {(x_i, v)}
20 retourner (π, θ)
```

Le solveur SCSP utilisé pour résoudre le réseau de contraintes stochastiques obtenu utilise une méthode de recherche arborescente avec retour arrière sur l'ensemble des valeurs du domaine de chaque variable. L'algorithme 1 illustre cette résolution. La méthode est appelée avec le seuil souhaité, une politique π initialement vide et l'indice de la première variable à assigner (initialement d'indice 0). Elle retourne une politique π qui satisfait les contraintes dures et tel que la probabilité de satisfaire les autres contraintes est supérieure ou égale au seuil θ_h. Notons que si le SCSP est insatisfiable, la politique retournée est vide. La fonction $isConsistent(x, v, C)$ retourne vrai si la valeur v pour la variable x satisfait l'ensemble des contraintes C. De même, la probabilité d'obtenir la valeur v pour la variable stochastique x est donnée par la fonction $prob(x, v)$.

Le tableau 4 indique les résultats obtenus sur le jeu du Verger allégé (deux fruits par arbre et un corbeau de taille 1) avec différents horizons et un seuil de 50%. Ces expérimentations ont été réalisées sur un bi-processeurs quad-core Intel XEON X5550 - 2,66 GHz - 8 Mo cache.

Horizon	Temps	#π
1	29 ms	0
2	28 s	0
3	4 min 53 s	0
4	37 min	6
5	4 h 54 min	12

TABLE 4 – Politiques gagnantes du Verger 2 2 1 avec $\theta \geq 50\%$

Les valeurs des variables $a_{bob,t}$ et $state_{0,t}$ à chaque temps dans les politiques satisfaisant le réseau de contraintes obtenu pour le jeu du Verger d'horizon 4 avec un seuil de 50% sont répertoriées dans le tableau 5. Les valeurs de $state_{0,t}$ sont des triplets représentants le nombre de fruits dans les arbres r et v et la taille du corbeau c. Les valeurs de la variable $a_{bob,t}$ correspondent à l'action « ne rien faire » (noop) et aux quatre actions de choix de fruits : (r,v), (v,r), (r,r), (v,v). Le joueur *random* lance le dé avec une probabilité équiprobable. On peut noter que les stratégies gagnantes sont celles faisant intervenir cette dernière action. C'est pourquoi, il est intéressant de noter qu'à $t = 1$ la quantité de fruits dans chaque arbre n'a pas été modifiée. Les stratégies optimales présentées dans le tableau à $t = 1$, nous indique que la stratégie optimale est de sélectionner un fruit dans chaque arbre. Les quatre premières solutions représentent le cas où le joueur *random* tire deux fois de suite le panier. Les solutions 5 et 6, le cas où le joueur a le choix au temps 1 puis le joueur *random* tire un fruit dans l'arbre v puis dans l'arbre r (solution 5) ou l'inverse (solution 6). Notons que le nombre de solutions est conditionné par la valeur de seuil. Avec un seuil de 50%, les solutions retournées correspondent à la stratégie optimale attendue dans le jeu du Verger.

6 Conclusion

Dans cet article, nous avons proposé de modéliser sous la forme de réseaux de contraintes stochastiques des jeux GDL à information incertaine et complète où un joueur représentant l'environnement joue indifféremment des actions des autres joueurs. La traduction de ce fragment de GDL utilise une sémantique basée sur un jeu de Markov. Nous avons montré que

Variable	t=1	t=2	t=3	t=4
$state_{0,t}$	2 2 1	1 1 1	1 1 1	0 0 1
$a_{bob,t}$	choice v r	noop	choice v r	undefined
$state_{0,t}$	2 2 1	1 1 1	1 1 1	0 0 1
$a_{bob,t}$	choice r v	noop	choice v r	undefined
$state_{0,t}$	2 2 1	1 1 1	1 1 1	0 0 1
$a_{bob,t}$	choice v r	noop	choice r v	undefined
$state_{0,t}$	2 2 1	1 1 1	1 1 1	0 0 1
$a_{bob,t}$	choice r v	noop	choice r v	undefined
$state_{0,t}$	2 2 1	1 1 1	1 0 1	0 0 1
$a_{bob,t}$	choice v r	noop	noop	undefined
$state_{0,t}$	2 2 1	1 1 1	0 1 1	0 0 1
$a_{bob,t}$	choice v r	noop	noop	undefined

TABLE 5 – Politiques gagnantes du Verger d'horizon 4 (où à l'état initial $state_{0,0}$ = (2 2 1) et $a_{bob,0}$ = noop)

notre réécriture est correcte par l'équivalence des modèles à chaque horizon. Nos premières expérimentations sur le jeu du Verger à l'aide d'un solveur SCSP générique sont concluantes et permettent de mettre en avant la stratégie optimale pour un horizon donné.

À court terme, nous souhaitons étendre notre modèle à un environnement dont les actions du joueur *random* peuvent dépendre de l'état courant. De même, des méthodes de filtrages adaptées au SCSP peuvent être utilisées afin d'accélérer la recherche de modèles. Plus généralement, ce travail offre de nombreuses perspectives dont notamment l'apprentissage automatique de stratégies optimales.

Références

[1] K. R. Apt, H. A. Blair, and A. Walker. Foundations of deductive databases and logic programming. chapter Towards a Theory of Declarative Knowledge, pages 89–148. Morgan Kaufmann, 1988.

[2] T. Balafoutis and K. Stergiou. Algorithms for stochastic CSPs. In *Proccedings of CP'06*, pages 44–58, 2006.

[3] C. Bessiere and G. Verger. Strategic constraint satisfaction problems. In *Proceedings of CP'06 Workshop on Modelling and Reformulation*, pages 17–29, 2006.

[4] N. Cesa-Bianchi and G. Lugosi. *Prediction, Learning, and Games*. Cambridge, 2006.

[5] J. E. Clune, III. *Heuristic evaluation functions for general game playing*. PhD thesis, University of California, Los Angeles, USA, 2008. Adviser-Korf, Richard E.

[6] E. Dantsin, T. Eiter, G. Gottlob, and A. Voronkov. Complexity and expressive power of logic programming. *ACM Comput. Surv.*, 33(3) :374–425, 2001.

[7] H. Finnsson and Y. Björnsson. Simulation-based approach to general game playing. In *Proc. of AAAI'08*, pages 259–264, 2008.

[8] M. Genesereth, N. Love, and B. Pell. General game playing : Overview of the aaai competition. *AAAI Magazine*, 26(2) :62–72, 2005.

[9] I. P. Genta, P. Nightingalea, A. Rowleya, and K. Stergiou. Solving quantified constraint satisfaction problems. *Artificial Intelligence*, 172(6-7) :738–77, 2008.

[10] V. Lifschitz and F. Yang. Eliminating function symbols from a nonmonotonic causal theory. In *Knowing, Reasoning, and Acting : Essays in Honour of Hector J. Levesque*. College Publicaions, 2011.

[11] I. W. Lloyd and R. W. Topor. A basis for deductive database systems. ii. *J. Log. Program.*, 30(1) :55–67, Apr. 1986.

[12] N. Love, T. Hinrichs, D. Haley, E. Schkufza, and M. Genesereth. General game playing : Game description language specification. Technical report, 2008.

[13] M. Möller, M. T. Schneider, M. Wegner, and T. Schaub. Centurio, a general game player : Parallel, Java- and ASP-based. *Künstliche Intelligenz*, 25(1) :17–24, 2011.

[14] T.-V.-A. Nguyen, A. Lallouet, and L. Bordeaux. Constraint games : Framework and local search solver. In *Proccedings of International Conference on Tools with Artificial Intelligence (ICTAI'13)*, pages 8–12, 2013.

[15] Y. Shoham and K. Leyton-Brown. *Multiagent Systems : Algorithmic, Game-Theoretic, and Logical Foundations*. Cambridge University Press, 2009.

[16] M. Thielscher. Flux : A logic programming method for reasoning agents. *Theory Pract. Log. Program.*, 5(4-5) :533–565, 2005.

[17] M. Thielscher. A general game description language for incomplete information games. In *Proc. of AAAI'10*, pages 994–999, 2010.

[18] T. Walsh. Stochastic constraint programming. In *Proc. of ECAI'02*, pages 111–115, 2002.

Une nouvelle caractérisation des intervalles d'intérêt pour le raisonnement énergétique

Alban Derrien

TASC (Mines Nantes, LINA, CNRS, INRIA),
4, Rue Alfred Kastler, FR-44307 Nantes Cedex 3, France.
alban.derrien@mines-nantes.fr

Résumé

Le raisonnement énergétique (ER) pour la contrainte Cumulative permet un filtrage des domaines supérieur à la plupart des algorithmes de l'état de l'art. Hélas, il est généralement trop coûteux pour être utilisé en pratique. Une des raisons de ce coût important est que trop d'intervalles de temps sont considérés. Dans la littérature, des approches heuristiques ont été développées dans le but de réduire le nombre d'intervalles à considérer, entrainant une perte de filtrage. Dans cet article, nous démontrons qu'il est possible de réduire le nombre d'intervalles d'intérêt jusqu'à un facteur sept pour la condition d'échec et le propagateur, sans diminuer la puissance de filtrage. En outre, nous montrons que, pour certaines classes de problèmes, associer ER à un algorithme de Time-Table constitue le meilleur compromis opérationnel.

1 Introduction

Les problèmes d'ordonnancement Cumulatifs (CuSP) sont présents dans de nombreux contextes industriels. Ils ont été très largement été étudiés en Programmation Par Contraintes (PPC). Le problème CuSP est défini par un ensemble d'activités $\mathcal{A}$ et une ressource de capacité C. C'est un problème NP-difficile. Chaque activité $a \in \mathcal{A}$ est définie par quatre variables : sa date de début de consommation s_a, sa durée p_a, sa date de fin e_a et sa consommation en ressource h_a. Nous utiliserons la notation $a = \{s_a, p_a, e_a, h_a\}$. Usuellement, les variables de durée et de consommation sont fixées. Une solution du CuSP est un ordonnancement de l'ensemble des taches vérifiant :

$$\forall a \in \mathcal{A} : s_a + p_a = e_a \quad \wedge \quad \forall i \in \mathbb{N} : \sum_{\substack{a \in \mathcal{A} \\ i \in [s_a, e_a[}} h_a \leq C$$

En PPC ce problème est généralement modélisé par la contrainte Cumulative [1].

Le raisonnement énergétique de Baptiste et al. (ER) est un des algorithmes de filtrage les plus puissants pour la contrainte Cumulative [2]. Cet algorithme utilise une caractérisation des intervalles dit *d'intérêt*, c'est-à-dire des intervalles suffisants pour vérifier que toutes les règles de filtrage sont satisfaites. Malheureusement, ER est souvent trop coûteux pour être utilisé en pratique. Tout d'abord, sa complexité en temps est $O(n^3)$. De plus, la constante cachée dans cette complexité en temps est élevée. En effet, de nombreux intervalles sont caractérisés comme étant d'intérêt alors que la plupart d'entre eux pourraient être ignorés. Dans la littérature, seules des approches heuristiques ont été proposées pour réduire ce nombre d'intervalles [3].

Cet article propose une caractérisation précise des intervalles d'intérêt pour la condition d'échec, que nous appelons *checker énergétique*, ainsi que pour le propagateur. Nous démontrons qu'il est possible de réduire ce nombre d'intervalles par un facteur sept pour le checker et pour le propagateur, sans perte de déduction. Nos expériences montrent une réduction non négligeable du temps d'exécution par rapport à l'existant. L'association d'un checker énergique et d'un algorithme de filtrage Time-Table (basé sur la base du profil cumulatif des parties obligatoires [7]) donne des résultats prometteurs. En outre, notre approche permet de répondre par l'affirmative à une question théorique ouverte [2] concernant les algorithmes de filtrage effectuant un raisonnement énergétique : les intervalles d'intérêt pour le checker énergétique sont suffisants pour réaliser un filtrage énergétique complet.

2 État de l'art

Pour une variable x, $\underline{x}$ représente la valeur minimum dans son domaine, et $\overline{x}$ la valeur maximale. Le raisonnement énergétique consiste à comparer l'énergie disponible

dans un intervalle (la durée de l'intervalle $\times$ la capacité C du problème) avec la quantité de ressource nécessairement consommée par les activités qui coupent cet intervalle. Pour une activité a, la longueur minimale occupée dans un intervalle $[t_1, t_2]$ est notée $MI(a, t_1, t_2)$:
$MI(a, t_1, t_2) = \max\big(0, \min\big(p_a, t_2 - t_1, \underline{e_a} - t_1, t_2 - \overline{s_a}\big)\big)$.

Proposition 1 (Checker énergétique [4]) *Si la condition*

$$\forall t_1, t_2 \in \mathbb{N}^2, t_1 < t_2$$

$$C \times (t_2 - t_1) \geq \sum_{a \in \mathcal{A}} h_a \times MI(a, t_1, t_2) \qquad (1)$$

n'est pas respectée, alors la contrainte Cumulative n'admet pas de solution.

La question est alors de trouver le plus petit ensemble d'intervalles $[t_1, t_2],\ t_2 > t_1$ devant être vérifiés pour détecter l'infaisabilité. À partir de cette condition une règle d'ajustement des bornes des domaines peut être définie : pour une activité a la borne $\underline{s_a}$ peut être mise à jour si l'activité l'ordonnancement de a à $\underline{s_a}$ entraîne nécessairement une surcharge (et de même concernant $\overline{s_a}$). L'étude du propagateur sera effectuée dans la section 4.

Caractérisation de Baptiste et al. Pour s'assurer que la condition (1) est respectée, Baptiste et al. [2] ont montré qu'il est suffisant de vérifier les intervalles de la forme suivante :

- $[t_1, t_2], t_1 \in O_1 < t_2 \in O_2$
- $[t_1, t_2], t_1 \in O_1 < t_2 \in O(t_1)$
- $[t_1, t_2], t_2 \in O_2 > t_1 \in O(t_2)$

avec $O_1 = \{\underline{s_a}, \forall a \in \mathcal{A}\} \cup \{\overline{s_a}, \forall a \in \mathcal{A}\} \cup \{\underline{e_a}, \forall a \in \mathcal{A}\}$, $O_2 = \{\overline{e_a}, \forall a \in \mathcal{A}\} \cup \{\overline{s_a}, \forall a \in \mathcal{A}\} \cup \{\underline{e_a}, \forall a \in \mathcal{A}\}$, $O(t) = \{\underline{s_a} + \overline{e_a} - t, \forall a \in \mathcal{A}\}$. Notons O_B l'ensemble de ces intervalles.

Cette caractérisation permet de réduire le nombre d'intervalles d'intérêt à $9+3+3=15$ pour chaque paire d'activité. Baptiste et al. ont démontré ([2], proposition 19) que cette caractérisation est suffisante pour détecter une surcharge en analysant la fonction représentant l'évolution de l'énergie disponible : $Slack(\mathcal{A}, t_1, t_2) = C \times (t_2 - t_1) - \sum_{a \in \mathcal{A}} h_a \times MI(a, t_1, t_2)$.

Sachant que $MI(a, t_1, t_2)$ peut être calculé en temps constant et compte tenu des définitions de O_1, O_2 et $O(t)$, nous obtenons un checker naïf en $O(n^3)$, en calculant $MI(a, t_1, t_2)$ pour tout t_1, t_2 et a. Cependant, la fonction de $Slack$ est linéaire et continue par morceaux. Un extremum local ne peut être trouvé que dans les points critiques de la fonction $Slack(\mathcal{A}, t_1, t_2)$. Ces points critiques ne peuvent exister que pour des valeurs de t_1 et t_2 correspondant aux intervalles caractérisés. Ces propriétés conduisent à un checker en $O(n^2)$ [2]. Deux questions subsistent, présentées comme ouvertes dans [2].

1. L'ensemble d'intervalles d'intérêt a été démontré comme étant suffisant, mais est il minimal en taille ?
2. Cet ensemble reste t-il suffisant pour appliquer un filtrage aux bornes ?

Caractérisation de Schwindt. Schwindt a proposé une caractérisation plus fine se basant sur une étude plus précise des extrémums de la fonction $f_1 : (t_1, t_2) \rightarrow Slack(\mathcal{A}, t_1, t_2)$ (théorèmes 3.7 et 3.8 dans [8], écrite en allemand). Cette fonction à deux variables est localement minimale seulement si sa dérivée à gauche est plus petite que sa dérivée à droite, à la fois pour t_1 et pour t_2. Comme $Slack(\mathcal{A}, t_1, t_2)$ est une somme d'intersections minimales, il doit exister une activité i (resp. j) telle que son intersection minimale a une dérivée à gauche plus grande que sa dérivée à droite sur t_1 (resp. t_2). Ces observations conduisent au théorème 1.

Théorème 1 *La fonction $Slack(\mathcal{A}, t_1, t_2)$ est localement minimum seulement si il existe deux activités i, j telles que les deux propositions suivantes sont satisfaites.*

$$\frac{\partial^- MI(i, t_1, t_2)}{\partial t_1} > \frac{\partial^+ MI(i, t_1, t_2)}{\partial t_1} \qquad (2)$$

$$\frac{\partial^- MI(j, t_1, t_2)}{\partial t_2} > \frac{\partial^+ MI(j, t_1, t_2)}{\partial t_2} \qquad (3)$$

Les valeurs de t_1 pour lesquelles la condition (2) est respectée sont appelées dates d'intérêt pour le début d'intervalle. De même, les valeurs de t_2 pour lesquelles la condition (3) est respectée sont appelées dates d'intérêt pour la fin d'intervalle. Leur conjonction donne alors les conditions nécessaires pour qu'un intervalle $[t_1, t_2[$ puisse être considéré comme d'intérêt pour le checker énergétique.

Schiwndt en a proposé une caractérisation en analysant les variations de la fonction d'intersection minimale, donnant 8 intervalles possibles, tous inclus dans la caractérisation de Baptiste et al. Cette caractérisation répond à la première question laissé ouverte : Le nombre d'intervalles dans la caractérisation de Baptiste et al. peut être réduit.

Nous proposons dans la section suivante une nouvelle analyse de la fonction d'intersection minimale, menant a une caractérisation plus précise des intervalles d'intérêt.

3 Caractérisation pour le checker

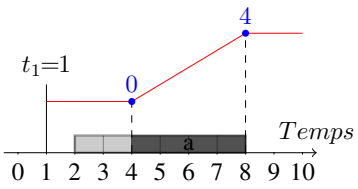
Dans cette section nous allons étudier l'inégalité (3) pour déterminer sous quelles conditions une date de fin peut être considérée comme étant d'intérêt. Le problème cumulatif étant symétrique, nous en déduirons de (2) les conditions pour l'intérêt d'une date de début.

Nous souhaitons caractériser, pour une date de début t_1 et une activité a les points d'intérêt issus de la fonction $f_2 : t_2 \rightarrow MI(a, t_1, t_2)$: Les valeurs de t_2 pour lesquelles la dérivée à gauche est plus grande que la dérivée à droite.

Propriété 1 *Pour une date de début d'intervalle t_1 et une activité a, la fonction $f_2 : t_2 \rightarrow MI(a, t_1, t_2)$ a au plus 1 point d'intérêt et ce point appartient à $O_2 \cup O(t_1)$.*

Preuve Nous démontrons la propriété 1 par une étude de cas : nous démontrons que pour les 4 positions de t_1 par rapport à a il existe au plus un point d'intérêt. Pour cela nous prouvons que la fonction f_2 est linéaire et continue par morceaux, composée d'au plus 3 morceaux. Les deux points critiques correspondent au début et à la fin de consommation. Nous démontrons ensuite que la fin de consommation est l'unique point avec une dérivée plus grande à gauche qu'à droite. Nous prenons un exemple graphique avec une même activité pour les quatre cas : $a = \{s_a \in [2,4],\ p_a = 4,\ e_a \in [6,8], h_a\}$.

1. $t_1 \leq \underline{s_a}$:



Nous rappelons la définition de $MI(a, t_1, t_2) = \max\left(0, \min\left(p_a,\ t_2 - t_1,\ \underline{e_a} - t_1,\ t_2 - \overline{s_a}\right)\right)$. Ce qui nous amène à étudier trois cas :

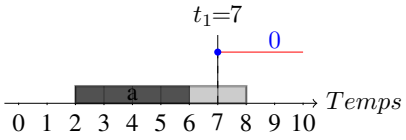
(a) si $t_2 \leq \overline{s_a}$ alors $MI(a, t_1, t_2) = 0$.

(b) si $\overline{s_a} \leq t_2 \leq \overline{e_a}$ alors $MI(a, t_1, t_2) = t_2 - \overline{s_a}$.

(c) si $\overline{e_a} \leq t_2$ alors $MI(a, t_1, t_2) = p_a$.

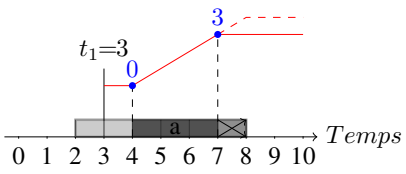
$p_a - (\overline{e_a} - t_2)$ est égale à 0 lorsque $t_2 = \overline{s_a}$ et p_a lorsque $t_2 = \overline{e_a}$. Donc lorsque $t_1 \leq \underline{s_a}$ la fonction de consommation est continue et linaire par morceaux, composée de trois morceaux, avec un unique point d'intérêt : $\overline{e_a}$.

2. $t_1 \geq \underline{e_a}$:



Dans ce cas $MI(a, t_1, t_2) = 0$ pour tout intervalle. La fonction est trivialement continue et linéaire par partie sans point d'intérêt.

3. $t_1 > \underline{s_a}$ et $t_1 < \underline{e_a}$ et $t_1 < \overline{s_a}$:



soit $\Delta = t_1 - \underline{s_a}$. Etudions trois cas pour les valeurs de t_2 :

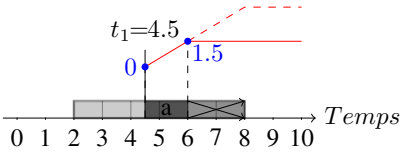
(a) si $t_2 \leq \overline{s_a}$ alors $MI(a, t_1, t_2) = 0$.

(b) si $\overline{s_a} \leq t_2 \leq \overline{e_a} - \Delta$ alors $MI(a, t_1, t_2) = t_2 - \overline{s_a}$.

(c) si $t_2 \geq \overline{e_a} - \Delta$ alors $MI(a, t_1, t_2) = p_a - \Delta$.

Dans chacun des trois cas, la fonction est linéaire. Puisque $t_2 - \overline{s_a}$ est égale à 0 lorsque $t_2 = \overline{s_a}$ et que $t_2 = \overline{s_a}$ et $p_a - \Delta$ lorsque $t_2 = \overline{e_a} - \Delta$, la fonction de consommation est continue et linéaire par morceaux, composée de 3 morceaux, avec un unique point d'intérêt : $\underline{s_a} + \overline{e_a} - t_1$.

4. $t_1 > \underline{s_a}$ et $t_1 < \underline{e_a}$ et $t_1 \geq \overline{s_a}$:



Nous avons ici deux cas distincts :

(a) si $t_2 \leq \underline{e_a}$ then $MI(a, t_1, t_2) = t_2 - t_1$

(b) si $MI(a, t_1, t_2) = p_a - \Delta$

Sachant que $t_2 - t_1 = p_a - \Delta$ lorsque $t_2 = \underline{e_a}$, la fonction de consommation est continue et linaire par morceaux, composée de deux parties, avec un unique point d'intérêt : $\underline{e_a}$.

Nous avons montré que quelque soit la date de début d'intervalle t_1 et quelque soit l'activité a, la fonction $f_2 : t_2 \rightarrow MI(a, t_1, t_2)$ est continue et linéaire par morceaux, avec au plus un point d'intérêt. Selon les cas les différents points d'intérêt sont : $\underline{e_a}$, $\underline{s_a} + \overline{e_a} - t_1$ et $\overline{e_a}$ appartenant tous les trois à $O_2 \cup O(t_1)$. Ce qui démontre la propriété. □

Nous avons caractérisé, pour une activité a, les valeurs possibles de fin d'intervalle pour lesquelles la condition (3) est vérifiée (Table 1). Par symétrie, nous pouvons déduire les valeurs de début d'intervalle pour lesquelles la condition (2) est vérifiée (Table 2).

conditions	t_2	
$t_1 \leq \underline{s_i}$	$\overline{e_i}$	cas 1
$t_1 > \underline{s_i} \ \wedge\ t_1 < \underline{e_i} \ \wedge\ t_1 < \overline{s_i}$	$\underline{s_i} + \overline{e_i} - t_1$	cas 3
$t_1 > \underline{s_i} \ \wedge\ t_1 < \underline{e_i} \ \wedge\ t_1 \geq \overline{s_i}$	$\underline{e_i}$	cas 4

TABLE 1 – Conditions d'intérêt de fin d'intervalle.

Le cas 3 et son symétrique, bien que satisfaisant les conditions nécessaires individuellement, ne peuvent amener ensemble à un intervalle d'intérêt. En effet, localement, la fonction $\delta \rightarrow MI(i, t_1 + \delta, t_2 - \delta)$ est égale à

conditions	intervalle	
$\underline{s_i} \leq \underline{s_j} \ \wedge \ \overline{e_j} \geq \overline{e_i}$	$[\underline{s_i},\ \overline{e_j}]$	A
$\underline{s_i} > \underline{s_j} \ \wedge \ \underline{s_i} < \underline{e_j} \ \wedge \ \underline{s_i} < \overline{s_j} \ \wedge \ \underline{s_j} + \overline{e_j} - \underline{s_i} \geq \overline{e_i}$	$[\underline{s_i},\ \underline{s_j} + \overline{e_j} - \underline{s_i}]$	B
$\underline{s_i} > \underline{s_j} \ \wedge \ \underline{s_i} < \underline{e_j} \ \wedge \ \underline{s_i} \geq \overline{s_j} \ \wedge \ \underline{e_j} \geq \overline{e_i}$	$[\underline{s_i},\ \underline{e_j}]$	C
$\overline{s_i} \leq \underline{s_j} \ \wedge \ \overline{e_j} < \overline{e_i} \ \wedge \ \overline{e_j} > \overline{s_i} \ \wedge \ \overline{e_j} \leq \underline{e_j}$	$[\overline{s_i}, \overline{e_j}]$	D
$\overline{s_i} > \underline{s_j} \ \wedge \ \overline{s_i} < \underline{e_j} \ \wedge \ \overline{s_i} < \overline{s_j} \ \wedge \ \overline{s_i} < \underline{s_j} + \overline{e_j} - \overline{s_i} \leq \underline{e_i} \ \wedge \ \underline{s_j} + \overline{e_j} - \overline{s_i} < \overline{e_i}$	$[\overline{s_i},\ \underline{s_j} + \overline{e_j} - \overline{s_i}]$	E
$\overline{s_i} > \underline{s_j} \ \wedge \ \overline{s_i} < \underline{e_j} \ \wedge \ \overline{s_i} \geq \overline{s_j} \ \wedge \ \underline{e_j} < \overline{e_i} \ \wedge \ \underline{e_j} > \overline{s_i} \ \wedge \ \underline{e_j} \leq \underline{e_i}$	$[\overline{s_i},\ \underline{e_j}]$	F
$\overline{e_j} < \overline{e_i} \ \wedge \ \overline{e_j} > \overline{s_i} \ \wedge \ \overline{e_j} > \underline{e_i} \ \wedge \ \underline{s_i} + \overline{e_i} - \overline{e_j} \leq \overline{s_j}$	$[\underline{s_i} + \overline{e_i} - \overline{e_j},\ \overline{e_j}]$	G
$\underline{e_j} < \overline{e_i} \ \wedge \ \underline{e_j} > \overline{s_i} \ \wedge \ \underline{e_j} > \underline{e_i} \ \wedge \ \overline{s_j} \leq \underline{s_i} + \overline{e_i} - \underline{e_j} < \underline{e_j} \ \wedge \ \underline{s_j} < \underline{s_i} + \overline{e_i} - \underline{e_j}$	$[\underline{s_i} + \overline{e_i} - \underline{e_j},\ \underline{e_j}]$	H

TABLE 3 – Intervalles d'intérêt pour une paire d'activités (i,j)

conditions	t_1	
$t_2 \geq \overline{e_i}$	$\underline{s_i}$	Sym. 1
$t_2 < \overline{e_i} \ \wedge \ t_2 > \overline{s_i} \ \wedge \ t_2 > \underline{e_i}$	$\underline{s_i} + \overline{e_i} - t_1$	Sym. 3
$t_2 < \overline{e_i} \ \wedge \ t_2 > \overline{s_i} \ \wedge \ t_2 \leq \underline{e_i}$	$\overline{s_i}$	Sym. 4

TABLE 2 – Conditions d'intérêt de début d'intervalle.

$MI(i, t_1, t_2) + \delta$. Nous ne sommes donc pas en présence d'un extremum pour l'intersection minimale. L'intervalle n'est donc pas d'intérêt pour trouver un minimum de la fonction de $Slack$.

Nous pouvons donc déduire 8 cas (de A à H) de la conjonction des conditions pour t_1 et t_2. La Table 3 résume les intervalles d'intérêt pour toute paire d'activités (i, j).

Notre caractérisation est plus précise que celle proposée par Schwindt : par exemple le cas *B* dans la Table 3 est plus précis que le cas équivalent *iii* dans la table 3.5 page 84 de [8]. Notons que pour tout couple d'activités seulement deux des cas sont possibles, les conditions étant mutuellement incompatibles. Nous avons donc réduit la caractérisation de 15 à 2 intervalles d'intérêt par couple d'activités.

De plus, nous pouvons remarquer que tout intervalle d'intérêt commence seulement aux dates de début au plus tôt et au plus tard $\underline{s_a}, \overline{s_a}$, ou alors finit à des valeurs de fins (plus tard ou plus tôt). Cette remarque amène à un allégement de l'algorithme en $O(n^2)$ pour le checker proposé par Baptiste et al. [2], en réduisant l'ensemble des dates de début d'intervalle O_1 à un nouvel ensemble : $O_S = \{\underline{s_a},\ \forall a \in \mathcal{A}\} \cup \{\overline{s_a}, \forall a \in \mathcal{A}\}$.

3.1 Nouvel algorithme pour le checker

Dans cette section nous décrivons un nouvel algorithme pour le checker énergétique, basé sur les principes de celui de Baptiste et al. [2], en limitant les dates de début d'intervalle à l'ensemble O_S tel que défini dans la section précédente ($O_S = \{\underline{s_a},\ \forall a \in \mathcal{A}\} \cup \{\overline{s_a}, \forall a \in \mathcal{A}\}$).

En triant les intervalles par date de fin croissante on peut calculer le changement de consommation entre deux intervalles, incrémentalement. Chaque début et fin de consommation sont représentés par un événement. Ces événements, triés par ordre de date croissante, appartiennent d'après la caractérisation précédente à quatre ensembles :

- $\mathcal{S}_M = (\overline{s_a}, a), \forall a \in \mathcal{A}$;
- $\mathcal{E}_m = (\underline{e_a}, a), \forall a \in \mathcal{A}$;
- $\mathcal{E}_M = (\overline{e_a}, a), \forall a \in \mathcal{A}$;
- $\mathcal{L}' = (\underline{s_a} + \overline{e_a} - t_1, a), \forall a \in \mathcal{A}$.

Nous avons vu dans la section précédente que le début de consommation d'énergie d'une activité (SoC_a) est soit au début de l'intervalle si le début d'intervalle intersecte la partie obligatoire de l'activité (cas 4), soit en $\overline{s_a}$ (cas 1&3). De plus, la fin de consommation d'énergie (EoC_a) est soit $\overline{e_a}$ si $t_1 \leq \underline{s_a}$ (cas 1), soit $\underline{e_a}$ si t_1 intersecte la partie obligatoire (cas 4), ou bien sinon $\underline{s_a} + \overline{e_a} - t_1$ si $t_1 > \underline{s_a}$ *et* $t_1 < \underline{e_a}$ *et* $t_1 < \overline{s_a}$ (cas 3).

```
1  foreach t_1 ∈ O_S do
2      pente = C − Σ_{a∈A} MI(a, t_1, t_1 + 1);
3      charge = 0 ;  t_2^old = t_1;
4      L' = {(t' − t_1, a) | (t', a) ∈ L};
5      foreach event(t_2, a) > t_1 in S_M, E_m, E_M, L' do
6          charge += pente × (t_2 − t_2^old) ;
7          if charge < 0 then Fail;
8          if event est un SoC_a then
9              pente −= h_a;
10         else if event est un EoC_a then
11             pente += h_a;
12         end
13         t_2^old = t_2;
14     end
15 end
```

Algorithm 1: ERC, un nouveau checker énergétique.

4 Caractérisation pour le propagateur

4.1 Suffisance des intervalles d'intérêt

Le raisonnement énergétique permet de filtrer les bornes des variables. Si l'ordonnancement d'une activité à son placement au plus tôt (placement le plus à gauche possible) induit une surcharge alors ce placement n'est pas valide et

la valeur $\underline{s_a}$ peut être filtrée. La consommation par décalage gauche de l'activité a qui coupe l'intervalle $[t_1, t_2[$ est définie par :

$$LS(a, t_1, t_2) = \max(0, \min(\underline{e_a}, t_2) - \max(\underline{s_a}, t_1))$$

Réciproquement la consommation par décalage à droite est :

$$RS(a, t_1, t_2) = \max(0, \min(\overline{e_a}, t_2) - \max(\overline{s_a}, t_1))$$

Par souci de lisibilité notons $Dispo(a, t_1, t_2)$ l'énergie disponible pour une activité a dans l'intervalle $[t_1, t_2[$:

$$Dispo(a, t_1, t_2) = Slack(\mathcal{A} \backslash a, t_1, t_2)$$

La règle de filtrage définie par Baptiste et al. s'écrit alors :

Proposition 2 *Pour une activité a si il existe un intervalle $[t_1, t_2[$ tel que :*

$$Dispo(a, t_1, t_2) \geq h_a \times LS(a, t_1, t_2) \quad (4)$$

n'est pas vérifiée alors le placement à gauche de l'activité a n'est pas valide ; et l'activité ne peut commencer avant :

$$t_2 - \frac{1}{h_a} \times Dispo(a, t_1, t_2) \quad (5)$$

Cette règle vérifie si l'activité a placée à son placement au plus tôt créée une surcharge. Si c'est le cas alors un filtrage de $\underline{s_a}$ peut être effectué en calculant la quantité d'énergie disponible dans l'intervalle pour l'activité a et en fixant sa date de début au plus tôt à ce moment là. Une règle symétrique utilisant le placement à droite $RS(a, t_1, t_2)$ permet de calculer une date après laquelle l'activité ne peut finir et d'effectuer un filtrage sur $\overline{e_a}$.

Proposition 3 *Pour une activité a si il existe un intervalle $[t_1, t_2[$ tel que :*

$$Dispo(a, t_1, t_2) \geq h_a \times RS(a, t_1, t_2) \quad (6)$$

n'est pas vérifiée alors, le placement à droite de l'activité a n'est pas valide ; et l'activité ne peut finir après :

$$t_1 + \frac{1}{h_a} \times Dispo(a, t_1, t_2) \quad (7)$$

Les règles de filtrage (4) et (6) s'appliquent à tout intervalle $[t_1, t_2[$. Il convient alors de trouver le plus petit ensemble d'intervalles devant être vérifiés pour détecter si un filtrage peut être effectué. Baptiste et al. proposent d'utiliser la caractérisation du checker pour le propagateur. Ils laissent malgré tout ouverte la question de complétude de cette caractérisation. Nous allons démontrer cette complétude puis nous en affinerons la caractérisation.

Théorème 2 $\forall [t_1, t_2[$, *(4) et (6) sont vérifiés.*

$$\Longleftrightarrow$$

$$\begin{cases} \forall [t_1, t_2[, t_1 \in O_1, t_2 \in O_2, & \textit{(4) et (6) sont vérifiés.} \\ \forall [t_1, t_2[, t_1 \in O_1, t_2 \in O(t_1), & \textit{(4) et (6) sont vérifiés.} \\ \forall [t_1, t_2[, t_2 \in O_2, t_1 \in O(t_2), & \textit{(4) et (6) sont vérifiés.} \end{cases}$$

Pour démontrer ce théorème commençons par démontrer l'équivalence pour la condition (4). De manière analogue à la démonstration précédente étudions les variations de la fonction associée :

$$f_3 : t_2 \rightarrow Dispo(a, t_1, t_2) - h_a \times LS(a, t_1, t_2) \quad (8)$$

Si toutes les valeurs de la fonction sont positives alors aucun filtrage n'est nécessaire d'après la règle de filtrage d'ER. Il suffit alors de chercher la présence de valeur négative. Démontrer le théorème revient alors à démontrer que tous les minimums de f_3 correspondent à des intervalles de O_B. Cette fonction étant la somme de fonctions linéaires et continues par morceaux les minimums locaux ne peuvent exister que si une partie de la somme est minimale.

L'étude des minimums induit par $Dispo(a, t_1, t_2)$ a déjà été effectuée (propriété 1) ; pour rappel $Dispo(a, t_1, t_2) = Slack(\mathcal{A} \setminus a, t_1, t_2)$. Il nous reste donc à caractériser les minimums de $-h_a \times LS(a, t_1, t_2)$. Cela revient à chercher les points d'intérêt de la fonction $f_4 : t_2 \rightarrow LS(a, t_1, t_2)$; les points pour lesquels la dérivée à gauche est plus petite que la dérivée à droite (théorème 1).

Propriété 2 *Pour une date de début d'intervalle t_1 et une activité a la fonction $f_4 : t_2 \rightarrow LS(a, t_1, t_2)$ a au plus 1 point d'intérêt : $\underline{e_a}$.*

Preuve En suivant le modèle de la démonstration de la propriété 1 nous étudions différents cas.

1. $t_1 \leq \underline{s_a}$:
   Alors $LS(a, t_1, t_2) = \max(0, \min(\underline{e_a}, t_2) - \underline{s_a})$ Etudions alors 3 sous cas :
   (a) si $t_2 \leq \underline{s_a}$ alors $LS(a, t_1, t_2) = 0$.
   (b) si $\underline{s_a} \leq t_2 \leq \underline{e_a}$ alors $LS(a, t_1, t_2) = t_2 - \underline{s_a}$.
   (c) si $\underline{e_a} \leq t_2$ alors $LS(a, t_1, t_2) = p_a$.
   Ce qui montre que lorsque $t_1 \leq \underline{s_a}$ alors la fonction de consommation est continue et linéaire par morceaux, composée de trois morceaux, avec seulement un point d'intérêt : $\underline{e_a}$.
2. $\underline{s_a} \leq t_1$:
   Alors $LS(a, t_1, t_2) = \max(0, \min(\underline{e_a}, t_2) - t_1)$ Etudions alors 2 sous cas :
   (a) si $t_2 \leq \underline{e_a}$ alors $LS(a, t_1, t_2) = t_2 - t_1$.
   (b) si $\underline{e_a} \leq t_2$ alors $LS(a, t_1, t_2) = \underline{e_a} - t_1$.
   Ce qui montre que lorsque $t_1 \geq \underline{s_a}$ alors la fonction de consommation est continue et linéaire par morceaux, composée de trois morceaux, avec seulement un point d'intérêt : $\underline{e_a}$.

Nous avons montré que quelque soit la date de début d'intervalle t_1 et quelque soit l'activité a la fonction f_4 ($t_2 \rightarrow LS(a, t_1, t_2)$) est continue et linéaire par morceaux, avec au plus un unique point d'intérêt : $\underline{e_a}$. Ce qui démontre la propriété. □

preuve du théorème 2 L'implication de la droite vers la gauche est évidente.

D'après les propriétés 1 et 2 les points d'intérêt de la fonction f_3 appartiennent à $O_2 \cup O(t_1)$. Par symétrie les points d'intérêt pour le début d'intervalle appartiennent à $O_1 \cup O(t_2)$. Les minimums de f_3 sont alors dans O_B. Ce qui démontre l'implication de la droite vers la gauche pour la condition (4).

La démonstration de l'implication pour la condition (6) est equivalente. □

Le théorème 2 répond directement à la seconde question ouverte :

Propriété 3 *La caractérisation proposée par Baptiste et al. est suffisante pour effectuer un filtrage énergétique complet.*

4.2 Caractérisation des intervalles d'intérêt

La table 3 donne la caractérisation des intervalles d'intérêt induits par $Slack(\mathcal{A}, t_1, t_2)$. Ces intervalles sont d'intérêt pour toutes les activités puisque la fonction $Slack$ est commune aux conditions de filtrage de chaque activité. Chaque activité a en outre des intervalles d'intérêt qui lui sont propres : ceux induit par $LS(a, t_1, t_2)$ et $RS(a, t_1, t_2)$. Dans le but d'obtenir une caractérisation fine des intervalles d'intérêt pour le propagateur nous nous intéressons maintenant à caractériser les intervalles définis par $LS(a, t_1, t_2)$ et $RS(a, t_1, t_2)$ qui n'ont pas déjà été caractérisés par $Slack(\mathcal{A}, t_1, t_2)$.

Par exemple la fonction de consommation à gauche $LS(a, t_1, t_2)$ nous donne comme date de fin d'intervalle d'intérêt $\underline{e_a}$. Or lorsque $t_1 \geq \overline{s_a}$ la fin d'intervalle $\underline{e_a}$ est déjà induite (cas 4). Les conditions d'intérêt de $\underline{e_a}$ pour le placement à gauche peuvent alors être renforcées : $t_1 < \underline{e_a} \wedge t_1 < \overline{s_a}$. Symétriquement les conditions d'intérêt pour $\underline{s_a}$ comme date de début sont : $t_2 > \underline{s_a} \wedge t_2 < \overline{e_a}$.

La fonction de consommation à droite nous donne comme date de fin d'intervalle d'intérêt $\overline{e_a}$. Or lorsque $t_1 \leq \underline{s_a}$ la fin d'intervalle $\overline{e_a}$ est déjà induite (cas 1). Les conditions d'intérêt pour $\overline{e_a}$ peuvent alors être renforcées : $t_1 < \overline{e_a} \wedge t_1 > \underline{s_a}$. Symétriquement les conditions d'intérêt pour $\overline{s_a}$ comme date de début sont : $t_2 > \overline{s_a} \wedge t_2 > \underline{e_a}$.

conditions	t_2	
$t_1 < \underline{e_i} \ \wedge \ t_1 < \overline{s_i}$	$\underline{e_i}$	LS
$t_1 < \overline{e_i} \ \wedge \ t_1 > \underline{s_i}$	$\overline{e_i}$	RS

TABLE 4 – conditions d'intérêt de fin d'intervalles

conditions	t_1	
$t_2 > \underline{s_i} \ \wedge \ t_2 < \overline{e_i}$	$\underline{s_i}$	Sym.LS
$t_2 > \overline{s_i} \ \wedge \ t_2 > \underline{e_i}$	$\overline{s_i}$	Sym.RS

TABLE 5 – conditions d'intérêt de début d'intervalles

De manière analogue à la construction des intervalles d'intérêt pour le checker nous pouvons établir que les intervalles d'intérêt pour le placement à gauche se construisent par la conjonction des conditions d'intérêt pour la date de début et la date de fin.
Pour une activité a les intervalles d'intérêt sont alors construit de trois manières :

1. Le début est d'intérêt pour $LS(a, t_1, t_2)$ et la fin est d'intérêt pour $Slack(i, t_1, t_2)$ pour une activité i.
2. Le début est d'intérêt pour $Slack(i, t_1, t_2)$ pour une activité i et la fin est d'intérêt pour $LS(a, t_1, t_2)$.
3. Le début et la fin sont d'intérêt pour $LS(a, t_1, t_2)$.

Puisqu'il existe seulement une date de fin (resp. début) d'intérêt par activité le cas 1 (resp. 2) induit $n-1$ intervalles d'intérêt pour l'activité a. Le cas 3 nous indique que l'intervalle $[\underline{s_a}, \underline{e_a}[$ est toujours d'intérêt pour le placement à gauche de l'activité a. Ces intervalles sont précisément caractérisés dans la table 6. Nous avons donc défini $2{\times}n{-}1$ intervalles d'intérêt pour le placement à gauche de l'activité a. De même pour le placement à droite (Table 7). Avec les $2{\times}n^2$ intervalles d'intérêt commun à chaque activité notre caractérisation permet de réduire le nombre d'intervalle d'intérêt pour chaque activité de $15n^2$ à $2n^2+4n-2$. Soit un facteur jusqu'à 7 lorsque le nombre d'activité est supérieur à 30 .

4.3 Nouvel algorithme pour le propagateur

Baptiste et al. proposent un algorithme calculant pour tous les intervalles d'intérêt l'énergie totale consommée puis vérifiant pour chaque activité si le placement à gauche (ou à droite) ne rajoute pas trop d'énergie, filtrant le cas échéant. Nous proposons dans notre nouvel algorithme de reprendre cette phase (ligne 1 à 16), en se limitant aux intervalles caractérisés dans la Table 3 (ce qui réduit le nombre d'intervalles calculés de $15n^2$ à $2n^2$).

```
1  foreach Intervalle d'intérêt [t1, t2[ (cf Table 3) do
2      W := Σ_{a∈A} h_a×MI(a, t1, t2);
3      if W > C × (t2 − t1) then
4          fail;
5      else
6          foreach activité a ∈ A do
7              Disp := C×(t2−t1) − W + h_a×MI(a, t1, t2);
8              if Disp < h_a.LS(a, t1, t2) then
9                  s_a := max(s_a, t2 − (1/h_a) × Disp);
10             end
11             if Disp < h_a.RS(a, t1, t2) then
12                 e_a := min(e_a, t1 + (1/h_a) × Disp);
13             end
14         end
15     end
16 end
```

conditions	intervalle	
$\underline{s_j} \leq \underline{s_i} \ \wedge \ \overline{e_i} < \overline{e_j}$	$[\underline{s_j},\ \overline{e_i}]$	LS_j1
$\underline{s_j} > \underline{s_i} \ \wedge \ \underline{s_j} < \underline{e_i} \ \wedge \ \underline{s_j} < \overline{s_i} \ \wedge \ \underline{s_i} + \overline{e_i} - \underline{e_j} > \underline{s_j} \ \wedge \ \underline{s_i} + \overline{e_i} - \underline{e_j} < \overline{e_j}$	$[\underline{s_j},\ \underline{s_i} + \overline{e_i} - \underline{e_j}]$	LS_j2
$\underline{s_j} > \underline{s_i} \ \wedge \ \underline{s_j} < \underline{e_i} \ \wedge \ \underline{s_j} \geq \overline{s_i} \ \wedge \ \overline{e_i} < \overline{e_j}$	$[\underline{s_j},\ \underline{e_i}]$	LS_j3
$\underline{s_j} < \overline{s_i} \ \wedge \ \overline{e_j} < \underline{e_i}$	$[\underline{s_j},\ \underline{e_i}]$	LS_i4
$\underline{s_j} + \overline{e_j} - \underline{e_i} < \underline{e_i} \ \wedge \ \underline{s_j} + \overline{e_j} - \underline{e_i} < \overline{s_i} \ \wedge \ \underline{e_i} < \overline{e_j} \ \wedge \ \underline{e_i} > \overline{s_j} \ \wedge \ \underline{e_i} > \underline{e_j}$	$[\underline{s_j} + \overline{e_j} - \underline{e_i},\ \underline{e_i}]$	LS_i5
$\overline{s_j} < \overline{s_i} \ \wedge \ \underline{e_i} < \overline{e_j} \ \wedge \ \underline{e_i} < \overline{s_j} \ \wedge \ \underline{e_i} \leq \underline{e_j}$	$[\overline{s_j},\ \underline{e_i}]$	LS_i6
	$[\underline{s_i},\ \underline{e_i}]$	LS_j7

TABLE 6 – Intervalles d'intérêt pour le placement à gauche.

conditions	intervalle	
$\underline{s_j} \leq \underline{s_i} \ \wedge \ \overline{e_i} > \underline{e_j}$	$[\overline{s_j},\ \overline{e_i}]$	RS_j1
$\overline{s_j} > \underline{s_i} \ \wedge \ \overline{s_j} < \underline{e_i} \ \wedge \ \overline{s_j} < \overline{s_i} \ \wedge \ \underline{s_i} + \overline{e_i} - \overline{s_j} < \overline{s_j} \ \wedge \ \underline{s_i} + \overline{e_i} - \overline{s_j} < \underline{e_j}$	$[\overline{s_j},\ \underline{s_i} + \overline{e_i} - \overline{s_j}]$	RS_j2
$\overline{s_j} > \underline{s_i} \ \wedge \ \overline{s_j} < \underline{e_i} \ \wedge \ \overline{s_j} \geq \overline{s_i} \ \wedge \ \underline{e_i} > \underline{e_j}$	$[\overline{s_j},\ \underline{e_i}]$	RS_j3
$\underline{s_j} > \underline{s_i} \ \wedge \ \overline{e_i} > \overline{e_j}$	$[\underline{s_j},\ \overline{e_i}]$	RS_i4
$\underline{s_j} + \overline{e_j} - \overline{e_i} < \overline{e_i} \ \wedge \ \underline{s_j} + \overline{e_j} - \overline{e_i} > \underline{s_i} \ \wedge \ \overline{e_i} < \underline{e_j} \ \wedge \ \overline{e_i} > \underline{e_j}$	$[\underline{s_j} + \overline{e_j} - \overline{e_i},\ \overline{e_i}]$	RS_i5
$\overline{s_j} > \underline{s_i} \ \wedge \ \overline{e_i} < \overline{e_j} \ \wedge \ \overline{e_i} > \overline{s_j} \ \wedge \ \overline{e_i} \leq \underline{e_j}$	$[\overline{s_j},\ \overline{e_i}]$	RS_i6
	$[\overline{s_i},\ \overline{e_i}]$	RS_i7

TABLE 7 – Intervalles d'intérêt pour le placement à droite.

L'algorithme doit ensuite calculer les intervalles d'intérêt pour chaque activité (il en existe $4n-2$).

foreach *activité* $a \in \mathcal{A}$ **do**
  foreach $[t_1, t_2[$ *d'intérêt pour* LS_a *(cf Table 6)* **do**
    $Disp := C \times (t_2 - t_1) - \sum_{i \in \mathcal{A} \setminus a} MI(i, t_1, t_2)$;
    if $Disp < h_a.LS(a, t_1, t_2)$ **then**
      $\underline{s_a} := \max(\underline{s_a},\ t_2 - \frac{1}{h_a} \times Disp)$;
    end
  end
  foreach $[t_1, t_2[$ *d'intérêt pour* RS_a *(cf Table 7)* **do**
    $Disp := C \times (t_2 - t_1) - \sum_{i \in \mathcal{A} \setminus a} MI(i, t_1, t_2)$;
    if $Disp < h_a.RS(a, t_1, t_2)$ **then**
      $\overline{e_a} := \min(\overline{e_a},\ t_1 + \frac{1}{h_a} \times Disp)$;
    end
  end
end

Algorithm 2: Propagateur énergétique.

5 Expérimentations

Nous avons effectué des expérimentations en Choco [9] version 3 (release 13.03), sur un ordinateur doté d'un processeur 2.9 GHz Intel Core i7. Nous avons effectué des tests sur des instances générées aléatoirement ainsi que sur des instances de la PSPLIB[6]. Les instances aléatoires ont 10 ou 20 activités, chacune ayant une durée choisie dans l'intervalle $[1, 10]$, et une hauteur dans $[1, 5]$. Nous avons utilisé une heuristique de branchement *first fail* [5].

5.1 Expérimentations du checker énergétique

Nous avons comparé trois algorithmes :
- L'algorithme 1, ERC ;
- L'algorithme de Baptiste et al. [2] ;
- L'algorithme naïf en $O(n^3)$ raisonnant sur l'ensemble des intervalles d'intérêt définis par la Table 3.

Le nombre de noeuds est identique pour toutes les instances, pour les trois algorithmes, comme nous l'attendions. La Table 10 montre un gain de temps d'exécution de l'ordre de 20 à 30% en utilisant l'algorithme 1 en comparaison à celui de Baptiste et al.

Instances	Algorithm 1 ($\mu s/node$)	Baptiste et al. ($\mu s/node$)	$O(n^3)$ ($\mu s/node$)
Random10	16.47	24.97	29.31
Random20	43.95	56.24	78.74
PspLib 30	450.67	618.77	1268.92
PspLib 120	1 339.24	1 683.26	11 288.54

TABLE 8 – Temps moyen par noeud

Nous avons aussi utilisé notre checker énergétique en combinaison avec l'algorithme de Time-Table de Letort et al. [7], en comparaison avec l'algorithme de propagation Time-Table Edge-Finding de Vilim [10]. Nous avons fixé une limite de temps à 5 minutes. De façon surprenante, sur les problèmes générés aléatoirement (mono ressource), le checker énergétique a pu dans le temps imparti prouver l'optimalité de 72 des 100 instances, alors que l'algorithme TTEF n'a pu en prouver que 7. Ces résultats montrent l'importance que peut avoir un checker énergétique dans un moteur de résolution de contraintes, dans certains contextes d'utilisation.

	TT	TT + TTEF	TT + Algorithm 1
Random20	6	7	72

TABLE 9 – #prouvé sur 100 instances.

5.2 Expérimentations du propagateur énergétique

Nous avons ensuite effectué des expérimentations du nouvel algorithme de propagation pour le raisonnement énergétique (Algorithme 2), en comparant deux algorithmes :

– L'algorithme 2 ;
– L'algorithme de Baptiste et al. [2] ;

Le nombre de noeuds est identique pour toutes les instances, conformément à ce que nous attendions. La Table 9 montre un temps d'exécution réduit d'un facteur 2 à 4 par l'utilisation de notre algorithme 2 en comparaison à celui de Baptiste et al.

Instances	Algorithm 2 ($\mu s/node$)	Baptiste et al. ($\mu s/node$)
Random10	91	244
Random20	327	641
PspLib 30	4 372	8 809
PspLib 120	41 418	151 390

TABLE 10 – Temps moyen par noeud

Nous avons aussi combiné notre propagateur énergétique avec l'algorithme de Time-Table, sur les mêmes 100 instances que celles présentées dans la Table 9. Cette combinaison a démontré l'optimalité pour 63 instances.

6 Conclusion et perspectives

Dans cet article nous avons proposé une nouvelle caractérisation des intervalles à considérer pour le checker et pour le propagateur énergétique. Notre caractérisation réduit le nombre d'intervalles d'intérêt par un facteur 7. De plus nous avons démontré que la caractérisation de Baptiste et al. est suffisante pour assurer un filtrage énergétique complet.

Nous avons de plus montré que l'utilisation de notre checker énergétique est le meilleur compromis pour prouver l'optimalité d'une certaine classe de problème.

Le filtrage énergétique est l'un des plus performant pour le problème cumulatif, ce papier montre qu'il est possible d'en améliorer le temps d'exécution sans perte de filtrage. Ceci ouvre des perspectives intéressantes pour obtenir un propagateur proposant un bon compromis filtrage-temps d'exécution.

Références

[1] Abder Aggoun and Nicolas Beldiceanu. Extending chip in order to solve complex scheduling and placement problems. *Math. Comput. Model.*, 17(7) :57–73, April 1993.

[2] Philippe Baptiste, Claude Le Pape, and Wim Nuijten. *Constraint-Based Scheduling : Applying Constraint Programming to Scheduling Problems*. International Series in Operations Research and Management Science. Kluwer, 2001.

[3] Timo Berthold, Stefan Heinz, and Jens Schulz. An approximative criterion for the potential of energetic reasoning. In *Proceedings of the First international ICST conference on Theory and practice of algorithms in (computer) systems*, TAPAS'11, pages 229–239, Berlin, Heidelberg, 2011. Springer-Verlag.

[4] Jacques Erschler, Pierre Lopez, and Catherine Thuriot. Scheduling under time and resource constraints. In *Proc. of Workshop on Manufacturing Scheduling, 11th IJCAI*, Detroit, USA, 1989.

[5] Robert M. Haralick and Gordon L. Elliot. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14(3) :263–313, 1980.

[6] Rainer Kolisch and Arno Sprecher. Psplib – a project scheduling problem library. *EUROPEAN JOURNAL OF OPERATIONAL RESEARCH*, 96 :205–216, 1996.

[7] Arnaud Letort, Nicolas Beldiceanu, and Mats Carlsson. A scalable sweep algorithm for the cumulative constraint. In Michela Milano, editor, *Principles and Practice of Constraint Programming*, Lecture Notes in Computer Science, pages 439–454. Springer Berlin Heidelberg, 2012.

[8] Christoph Schwindt. *Verfahren zur Lösung des ressourcenbeschränkten Projektdauerminimierungsproblems mit planungsabhängigen Zeitfenstern*. PhD thesis, Fakultät für wirtschaftswissenschaften der Universität Fridericiana zu Karlsruhe, 1998. in German.

[9] CHOCO Team. Choco : an open source Java CP library. Research report 10-02-INFO, Ecole des Mines de Nantes, 2010.

[10] Petr Vilím. Timetable edge finding filtering algorithm for discrete cumulative resources. In Tobias Achterberg and J. Beck, editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, volume 6697 of *Lecture Notes in Computer Science*, pages 230–245. Springer Berlin / Heidelberg, 2011.

Un nouveau paramètre de graphes pour la résolution de CSP par décomposition *

Philippe Jégou **Cyril Terrioux**
Aix-Marseille Université, LSIS UMR 7296
Avenue Escadrille Normandie-Niemen
13397 Marseille Cedex 20 (France)
{philippe.jegou, cyril.terrioux}@lsis.org

Résumé

D'un point de vue théorique, les méthodes de décompositions (arborescentes) offrent une bonne approche lorsque la largeur (arborescente) des réseaux de contraintes (CSP) est petite. Dans ce cas, elles ont souvent montré leur intérêt pratique. Ainsi, la littérature, en provenance des Mathématiques ou de l'Intelligence Artificielle, a concentré l'essentiel de ses efforts à minimiser un seul paramètre, la largeur arborescente (ou tree-width). Néanmoins, des études expérimentales ont montré que ce paramètre n'est pas toujours le plus pertinent à considérer pour la résolution de CSP.

En particulier, dans cet article, nous montrons expérimentalement que les algorithmes de décomposition de l'état de l'art produisent des clusters (une décomposition arborescente est un arbre de clusters) ayant plusieurs composantes connexes. Ensuite, nous mettons en évidence que la présence de tels clusters crée un réel problème pour l'efficacité des méthodes de résolution. Pour éviter ce genre de problème, nous présentons ici un nouveau paramètre graphique dans le cadre des CSP, et qui est appelé la *Bag-Connected Tree-Width*, qui ne tient compte que des décompositions arborescentes dont chaque cluster est connexe, et qui sont appelées *Bag-Connected Tree-Decompositions*. Un premier algorithme polynomial calculant ces décompositions est proposé. Enfin, nous montrons expérimentalement que l'utilisation de ces Bag-Connected Tree-Decompositions améliore considérablement la résolution de CSP par les méthodes de décomposition.

*Ce travail est soutenu par l'Agence Nationale de la Recherche dans le cadre du projet TUPLES (ANR-2010-BLAN-0210).

1 Introduction

Les Problèmes de Satisfaction Contraintes (CSP) offrent un moyen assez puissant pour la formulation de problèmes en informatique, et en particulier en Intelligence Artificielle. Formellement, un *Problèmes de Satisfaction Contraintes* est un triplet (X, D, C), où $X = \{x_1, \ldots, x_n\}$ est un ensemble de n variables, $D = (D_{x_1}, \ldots, D_{x_n})$ est un ensemble de domaines finis de valeurs, un par variable, et $C = \{C_1, \ldots, C_e\}$ est un ensemble fini de e contraintes. Chaque contrainte C_i est une paire $(S(C_i), R(C_i))$, où $S(C_i) = \{x_{i_1}, \ldots, x_{i_k}\} \subseteq X$ définit la *portée* ou *scope* de C_i, et $R(C_i) \subseteq D_{x_{i_1}} \times \cdots \times D_{x_{i_k}}$ est sa *relation de compatibilité*. L'*arité* de C_i est notée $|S(C_i)|$. Un CSP est dit *binaire* si toutes ses contraintes sont d'arité 2. La structure d'un réseau de contraintes (autre terme pour appeler un CSP) est représentée par un hypergraphe (qui est un graphe dans le cas binaire), appelé *(hyper)graphe de contraintes*, et dont les sommets correspondent aux variables et les arêtes aux portées des contraintes. Dans cet article, pour une question de simplification, nous n'évoquerons que le cas des CSP binaires mais ce travail peut directement être étendu aux CSP non binaires en s'appuyant sur la *2-section* [2] de l'hypergraphe de contraintes (aussi appelée *graphe primal*), comme cela sera fait dans la partie expérimentale puisque nous y traiterons à la fois des CSP binaires et non binaires. De plus, et sans manque de généralité, nous supposerons que les réseaux considérés sont connexes. Pour simplifier les notations, dans la suite, nous noterons le graphe $(X, \{S(C_1), \ldots S(C_e)\})$ par (X, C). Une affectation sur un sous-ensemble de X sera dite *cohérente* si elle ne viole aucune contrainte. Vérifier si un CSP possède une *solution* (i.e. une affectation cohérente de toutes

les variables) est bien connu pour constituer un problème NP-complet. Aussi, de nombreux travaux ont été développés pour rendre la résolution d'instances très efficace en pratique, cela, en utilisant des formes optimisées de backtracking, des heuristiques, de l'apprentissage de contraintes, des techniques de backtracking non-chronologique, des techniques de filtrage basées sur la propagation de contraintes, etc. La complexité de ces approches demeure bien évidemment exponentielle, au moins en $O(n.d^n)$ où n est le nombre de variables et d la taille maximum des domaines.

Une autre approche de la problématique concerne l'étude des classes polynomiales qui s'appuient sur des propriétés des réseaux de contraintes. Il a notamment été montré que si la structure d'un CSP binaire est acyclique, il peut être résolu en temps linéaire [10]. En exploitant et en généralisant ces résultats théoriques, des méthodes de résolution de CSP ont été définies, comme par exemple le *Tree-Clustering* [7]. Ce type de méthodes s'appuie sur la notion de *décomposition arborescente* (*Tree-Decomposition*) de graphes [19]. Leur avantage fondamental est lié à leur complexité théorique, de l'ordre de d^{w+1} où w est la *largeur arborescente* (*Tree-Width*) du graphe de contraintes. Quand ce graphe possède de « belles » propriétés topologiques, à savoir quant w est « petit », ces méthodes permettent de résoudre des instances de grande taille. C'est le cas par exemple des problèmes bien connus d'allocation de fréquences radio [4]. Notons qu'en pratique, la complexité en temps est plutôt de l'ordre de $d^{w^+ +1}$ où $w^+ \geq w$ est en fait une approximation de la largeur arborescente car le calcul de décompositions arborescentes optimales (i.e. de largeur w) constitue un problème NP-difficile [1].

Toutefois, la mise mise en œuvre pratique de ce type de méthodes, bien qu'elle ait très souvent démontré son intérêt, a également permis de constater que la minimisation du paramètre w^+ n'est pas toujours la plus appropriée. Outre la difficulté du calcul de la valeur optimale de w^+, i.e. w, la manipulation de décompositions optimales ne conduit pas toujours à une résolution des plus efficaces en pratique. Cela a d'ailleurs mené à proposer des méthodes de décomposition de graphes qui rendent la résolution de CSP plus efficace (d'un point de vue pratique), mais pour lesquelles w^+ peut être significativement plus grand que w [14].

Dans cette contribution, nous montrons que l'une des raisons à ce manque d'efficacité dans la résolution de CSP par décomposition peut se trouver dans la nature même des décompositions pour lesquelles w^+ est proche de w. En effet, la minimisation de w^+ peut conduire à des décompositions pour lesquelles certains clusters recèlent plusieurs composantes connexes. Malheureusement, cette absence de connectivité conduit à des méthodes de résolution qui vont dépenser un temps considérable dans la résolution de sous-problèmes relatifs à ces clusters déconnectés, en passant beaucoup de temps à aller d'une composante connexe à une autre. Pour éviter ce problème, nous introduisons dans le cadre des CSP un nouvel invariant de graphe avec un paramètre appelé *Bag-Connected Tree-Width*. Ce paramètre est égal à la largeur minimale pour toutes les décompositions arborescentes pour lesquelles chaque cluster possède une unique composante connexe. Ces décompositions seront appelées *Bag-Connected Tree-Decompositions*. Ce paramètre qui a été introduit très récemment dans [18] [1] est égal à la largeur minimum pour toutes les Bag-Connected Tree-Decompositions. Nous démontrons ici que le calcul de ce paramètre est NP-difficile. Aussi, nous proposons un premier algorithme de complexité polynomiale en temps, précisément $O(n(n+e))$, dont l'objet est d'approximer ce paramètre à l'aide d'un calcul de décomposition associée. Les expérimentations que nous présentons montrent la pertinence de ce paramètre puisque sa prise en compte permet d'améliorer significativement la résolution de CSP par décomposition.

Notons que ce travail s'appuie sur la décomposition arborescente, mais qu'il pourrait très bien être adapté à d'autres méthodes de décompositions (par exemple [12, 13]). En effet, pour la plupart des méthodes de résolution de CSP à base de décomposition, les décompositions sont en fait calculées à l'aide d'algorithmes qui visent à approximer au mieux un paramètre graphique (la largeur) sans prendre en compte ni la connectivité des clusters produits, ni d'ailleurs la phase de résolution qui va suivre. Aussi, les problèmes observés ici avec la décomposition arborescente peuvent également se présenter pour d'autres formes de décompositions.

La partie suivante introduit les principes des méthodes de résolution de CSP par décomposition arborescente. La partie 3 met en évidence certains des problèmes inhérents au calcul de « bonnes » décompositions tandis que la partie 4 introduit la notion de Bag-Connected Tree-Decomposition, tout en proposant un premier algorithme qui permet d'en calculer. Avant de conclure, nous présentons des résultats expérimentaux qui montrent l'intérêt que recèle l'exploitation de cette notion pour la résolution pratique de CSP.

2 Résolution de CSP par décomposition

Le *Tree-Clustering* (noté *TC* [7]) constitue la méthode de référence pour la résolution de CSP binaires

1. Au moment de la soumission, nous ignorions l'existence de cette nouvelle notion essentiellement étudiée d'un point de vue combinatoire car elle n'apparaît actuellement dans aucune publication mais uniquement dans un rapport technique.

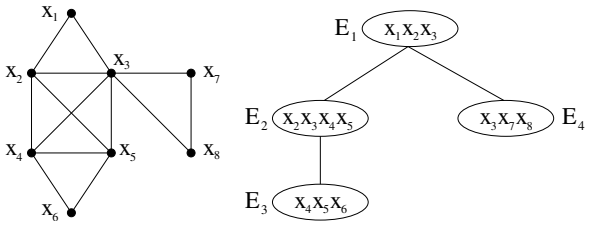


FIGURE 1 – Un graphe de contraintes de 8 variables (a) et une décomposition arborescente optimale (b).

via l'exploitation de la structure de leur graphe de contraintes. Cette méthode est basée sur la notion de décomposition arborescente de graphes [19].

Définition 1 *Étant donné un graphe $G = (X, C)$, une* décomposition arborescente *de G est une paire (E, T) où $T = (I, F)$ est un arbre et $E = \{E_i : i \in I\}$ une famille de sous-ensembles de X, telle que chaque sous-ensemble (appelé cluster ou bag au niveau mathématique) E_i est un nœud de T et vérifie :*

(i) $\cup_{i \in I} E_i = X$,

(ii) pour chaque arête $\{x, y\} \in C$, il existe $i \in I$ avec $\{x, y\} \subseteq E_i$, et

(iii) pour tout $i, j, k \in I$, si k est un chemin de i vers j dans T, alors $E_i \cap E_j \subseteq E_k$.

Notons que la condition (iii) peut être remplacée par : si un sommet x est tel que $x \in E_i \cap E_j$, alors, tous les nœuds E_k de T qui apparaissent dans l'unique chemin de E_i à E_j contiennent x. La largeur *d'une décomposition arborescente (E, T) est égale à $max_{i \in I}|E_i| - 1$. La* largeur arborescente *w de G est la largeur minimale pour toutes les décompositions arborescentes de G.*

La figure 2(b) présente un arbre dont les nœuds correspondent aux cliques maximales du graphe présenté dans la figure 2(a). Il s'agit de l'une des décompositions arborescentes de ce graphe. Ainsi, nous avons $E_1 = \{x_1, x_2, x_3\}$, $E_2 = \{x_2, x_3, x_4, x_5\}$, $E_3 = \{x_4, x_5, x_6\}$, et $E_4 = \{x_3, x_7, x_8\}$. La taille maximum des clusters dans cette décomposition arborescente optimale vaut 4 et donc, la largeur arborescente de ce graphe vaut 3.

La première version de TC [7] débute par le calcul d'une décomposition arborescente (qui utilise l'algorithme MCS [22]). Dans la seconde étape, les clusters sont résolus indépendamment, en considérant chaque cluster comme un sous-problème, et donc, en énumérant toutes ses solutions. Après cela, une solution globale au CSP, s'il en existe une, peut alors être efficacement calculée en exploitant la structure arborescente de la décomposition. Les complexités en temps et en espace de cette première version sont en $O(n.w^+.log(d).d^{w^+ + 1})$ où $w^+ + 1$ est la taille du plus grand cluster ($w + 1 \leq w^+ + 1 \leq n$). Notons que cette première approche a été améliorée pour arriver à une complexité en espace en $O(n.s.d^s)$ [6, 5] où s est la taille de la plus grande intersection (*séparateur*) entre deux clusters ($s \leq w^+$). Malheureusement, ce type d'approche qui résout complètement chaque cluster n'est pas efficace dans la pratique. Afin de contourner ce type de problème, la méthode *BTD* (pour *Backtracking on Tree-Decomposition* [16]) a été proposée et a finalement démontré tout son intérêt sur le plan pratique, et figure désormais comme une méthode de référence dans l'état de l'art pour ce type d'approches. Contrairement à TC, BTD n'a pas besoin de résoudre exhaustivement chaque cluster pour démontrer l'existence de solution. Une recherche de type backtrack est réalisée en exploitant un ordre sur les variables, induit par une recherche en profondeur dans la décomposition arborescente considérée. Alors que cette approche a démontré son intérêt pratique, d'un point de vue théorique, dans le pire des cas, ses complexités en temps et en espace sont du même ordre que celles des versions améliorées de TC, précisément $O(n.s^2.e.log(d^s).d^{w^+ + 1})$ pour la complexité en temps, et $O(n.s.d^s)$ pour la complexité en espace. Aussi, pour rendre une méthode structurelle efficace, il faut *a priori* minimiser les valeurs de w^+ et s lors du calcul de la décomposition arborescente. Malheureusement, calculer une décomposition arborescente optimale (i.e. de largeur w) est NP-difficile [1]. Aussi, de très nombreux travaux ont abordé cette question. Ils exploitent souvent une approche algorithmique fondée sur la notion de graphe *triangulé* (se reporter à [11] pour une introduction à cette classe de graphes). Nous pouvons distinguer différentes classes d'approches. Tout d'abord, il y a les méthodes recherchant des décompositions optimales ou bien leur approximation (avec garanties) mais qui n'ont pas démontré à ce jour leur intérêt pratique pour une raison de temps d'exécution extrêmement important au regard de la faible amélioration de la valeur de w^+ qu'elles permettent d'obtenir. Viennent ensuite les méthodes n'offrant aucune garantie en termes d'optimalité (comme celles fondées sur les *triangulations heuristiques*) mais qui s'avèrent les plus utilisées au niveau pratique. Elles opèrent en temps polynomial (entre $O(n + e)$ et $O(n^3)$), sont très simples à implémenter, et leur avantage semble tout à fait justifié. En effet, ces heuristiques semblent produire des triangulations assez proches de l'optimum [17]. En pratique, les méthodes les plus usitées pour calculer des décompositions arborescentes sont basées sur MCS [22] et Min-Fill [20] qui offrent de bonnes approximations de w^+. De plus, dans [14], les expérimentations ont montré que l'efficacité pour la résolution de CSP n'est pas seulement corrélée à la valeur de w^+, mais aussi à celle de s. Néanmoins, à notre connaissance,

ces études se sont essentiellement concentrées sur la valeur de w^+ et à un degré moindre de s, mais pas sur la structure interne des clusters qui semble également fournir un paramètre des plus pertinents. Cette question est développée dans la partie 3 qui montre que les propriétés topologiques des clusters constituent également un paramètre crucial pour la résolution de CSP.

3 Impact des clusters non connexes

L'étude des décompositions arborescentes montre qu'elles recèlent très souvent des clusters constitués de plusieurs composantes connexes. Par exemple, considérons un réseau constitué d'un cycle sans corde (c'est-à-dire sans arête joignant deux sommets non consécutifs dans le cycle) de n sommets (avec $n \geq 4$). Toute décomposition arborescente optimale possède exactement $n-2$ clusters de taille 3, et parmi eux, $n-4$ clusters possèdent 2 composantes connexes.

Ce phénomène est également observé sur des instances réelles, dès lors que l'on considère des décompositions arborescentes de qualité satisfaisante. Par exemple, la célèbre instance RLFAP *Scen-06* que l'on retrouve notamment dans la compétition CSP 2008 [2] est définie sur 200 variables et son réseau admet des décompositions arborescentes de petite largeur et qui peuvent être calculées très facilement (e.g. Min-Fill en trouve une pour laquelle $w^+ = 20$). Malheureusement, une analyse détaillée de ces décompositions arborescentes montre qu'elles possèdent plusieurs clusters non connexes. Plus généralement, il s'avère qu'environ 32% des 7272 instances de la compétition CSP 2008 possèdent des décompositions arborescentes qui recèlent au moins un cluster non connexe quand MCS ou Min-Fill sont utilisées, ce qui est généralement le cas quand on emploie ce type de méthodes pour la résolution de CSP. Parmi ces instances pour lesquelles MCS ou Min-Fill produisent de telles décompositions arborescentes, nous retrouvons notamment la plupart des instances RLFAP ou FAPP qui se trouvent être par ailleurs souvent exploitées comme benchmarks pour ce type de méthodes, à la fois pour les problèmes de décision et pour ceux d'optimisation. De plus, parfois, le pourcentage de clusters non connexes dans ces instances peut être considérable, atteignant jusqu'à 99% et en moyenne aux environs de 35%. Pour les instances FAPP, la moyenne est d'environ 48% pour les décompositions arborescentes produites par Min-Fill, et plus importante encore en utilisant MCS. Cette observation serait encore plus frappante pour les algorithmes qui trouvent des décompositions de largeurs plus petites, comme illustré par l'exemple du cycle sans corde.

La présence de clusters non connexes dans les décompositions arborescentes considérées peut avoir un impact extrêmement négatif sur l'efficacité pratique des méthodes de décomposition qui seront alors sanctionnées, lors de la résolution des instances, par un temps de calcul très important et un recours considérable à la mémoire. Pour bien comprendre cela, il faut déjà se rappeler que si un réseau de contraintes n'est pas connexe, cela peut avoir des conséquences importantes sur l'efficacité de sa résolution. Par exemple, si l'une de ses composantes connexes n'a pas de solution, et si la résolution aborde d'abord une composante connexe qui en possède, chacune de ses solutions devra être obtenue avant de prouver l'incohérence du CSP. Pour le cas des méthodes de décomposition, l'existence de clusters non connexes est peut-être encore plus pernicieuse. Dans le cas de TC, considérons un cluster non connexe. D'une part, le phénomène déjà rencontré dans le cas de réseaux non connexes peut se présenter. Mais il est également possible que ce cluster possède des solutions. Toutes ces solutions seront alors calculées et mémorisées avant de traiter un autre cluster. Leur nombre peut être considérable car il correspond au produit du nombre de solutions de chacune de ses composantes connexes. Notons de plus que pour certains benchmarks provenant des instances FAPP, le nombre de composantes connexes dans un cluster peut même être supérieur à 100. Toutefois, de nombreuses solutions locales à ce cluster peuvent être incompatibles globalement, car ces composantes connexes seront en général reliées par des contraintes qui apparaissent dans d'autres clusters. La figure 2 présente un exemple de décomposition pour laquelle deux composantes connexes d'un cluster E_i sont connectées via une séquence de contraintes qui apparaissent dans un sous-problème enraciné dans ce cluster. Ainsi, l'incohérence globale des solutions locales de E_i ne peut être détectée que quand ces clusters auront été résolus, au moment de la recomposition de solutions globales produites par TC lors de sa dernière étape. Cela conduit TC à une consommation considérable de temps et de mémoire, rendant de fait cette approche irréaliste en pratique.

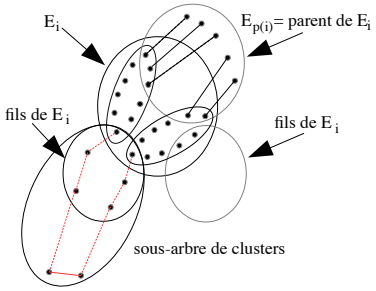


FIGURE 2 – Cluster non connexe au sein d'une décomposition arborescente.

D'autres méthodes ont été proposées de sorte à éviter ce type de phénomène où les clusters sont résolus indépendamment lors d'une première étape. C'est no-

2. Voir http ://www.cril.univ-artois.fr/CPAI08.

tamment le cas de BTD qui est l'une des approches les plus efficaces basées sur des décompositions. Bien que BTD ait montré son intérêt pratique, le phénomène observé existe toujours, même s'il s'avère généralement atténué. Pour bien comprendre cela, nous devons rappeler que BTD résout une instance en résolvant successivement les sous-problèmes enracinés dans chaque cluster de la décomposition arborescente. Mais contrairement à TC qui calcule d'abord toutes les solutions d'un cluster, lors de l'accès à un cluster, BTD se limite à ne calculer qu'une seule solution. Grossièrement, le sous-problème enraciné dans un cluster E_i correspond à un sous-problème impliquant toutes les variables figurant dans la descendance de E_i dans la décomposition arborescente (voir [16] pour plus de détails). BTD commence une recherche de type backtrack en affectant systématiquement les variables du cluster racine avant d'explorer un cluster fils. Lors de l'exploration d'un nouveau cluster E_i, BTD affecte les variables qui apparaissent dans le cluster E_i exceptées celles figurant dans le séparateur $E_i \cap E_{p(i)}$ [3]. Par exemple, considérons le graphe de contraintes de la figure 2 et sa décomposition arborescente associée. Si nous supposons que E_1 est le cluster racine, BTD essaye d'abord d'affecter de façon cohérente les variables de E_1. Si c'est le cas, BTD poursuit la recherche avec l'un des clusters fils (i.e. E_2 ou E_4). Si BTD choisit d'explorer d'abord E_2, il devra affecter de façon cohérente les variables de $E_2 \backslash (E_1 \cap E_2)$ (i.e x_4 et x_5).

Maintenant et plus généralement, considérons le cas d'un cluster non connexe E_i. Nous avons deux cas :

- si $G[E_i \backslash (E_i \cap E_{p(i)})]$ [4] est déconnecté : BTD doit affecter de façon cohérente les variables qui sont réparties dans plusieurs composantes connexes. Si le sous-problème enraciné en E_i est trivialement cohérent (par exemple, il admet un grand nombre de solutions), BTD va trouver une solution en faisant au plus quelques backtracks et enchaîner directement sur la recherche dans le cluster suivant. Ainsi, dans un tel cas, la non-connexité de E_i n'entraîne pas de problème. En revanche, si ce sous-problème a peu de solutions, voire aucune, nous avons une probabilité significative que BTD passe beaucoup de temps d'une composante connexe de $G[E_i \backslash (E_i \cap E_{p(i)})]$ à une autre lorsqu'il résoudra ce cluster. BTD peut avoir à explorer toutes les affectations cohérentes de chaque composante connexe en intercalant éventuellement les variables des différentes composantes connexes. En effet, si BTD exploite des techniques de filtrage, l'affectation d'une valeur à une variable x de $E_i \backslash (E_i \cap E_{p(i)})$ a une incidence principalement sur les variables de la composante connexe de $G[E_i \backslash (E_i \cap E_{p(i)})]$ qui contient x. En revanche, le filtrage ne modifiera pas ou très peu le domaine d'une variable figurant dans une autre composante connexe. Cela implique que les incohérences seront souvent détectées plus tard et pas nécessairement dans E_i mais dans l'un de ses clusters descendants (comme illustré dans la figure 2). Si c'est le cas, BTD peut nécessiter une quantité considérable de temps et de mémoire (à cause de l'enregistrement de (no-)goods) pour résoudre le sous-problème enraciné en E_i, surtout si les variables ont des domaines de grande taille. Ce phénomène négatif a par exemple été observé empiriquement sur certains benchmarks de la classe FAPP (e.g. l'instance *normalized-fapp05-0350-10*) avec une version de BTD basée sur MAC [21].
- si $G[E_i \backslash (E_i \cap E_{p(i)})]$ est connexe : nécessairement, E_i est un cluster non connexe parce que son séparateur avec son cluster père n'est pas connexe. Comme les variables de ce séparateur sont déjà affectées, la non-connectivité de E_i n'entraîne aucun problème.

Cet impact négatif des clusters non connexes est tout à fait compatible avec les résultats empiriques rapportés dans la littérature. Nous avons constaté que parfois, le pourcentage de clusters non connexes avec Min-Fill diffère sensiblement d'avec celui de MCS, ce qui pourrait notamment expliquer certaines différences d'efficacité observées par différents auteurs. En effet, même si la largeur est identique, les décompositions calculées par Min-Fill offrent de meilleurs résultats pour la résolution que ceux obtenus avec MCS [14] ; et Min-Fill est considérée comme étant la meilleure heuristique de l'état de l'art. Par ailleurs, l'analyse des décompositions arborescentes montre également que la connexion entre composantes connexes de certains clusters est fréquemment observée seulement au niveau des clusters feuilles de la décomposition, ce qui augmente davantage les effets négatifs observés. Pour éviter ce genre de phénomène, nous étudions dans la partie 4 les classes de décompositions arborescentes pour lesquelles tous les clusters devront être connexes.

4 Un nouveau paramètre pour la décomposition de réseaux de contraintes

Les constats présentés plus haut nous conduisent tout naturellement à ne considérer que des décompositions arborescentes pour lesquelles, les clusters seraient tous connexes. Cette notion a très récemment été introduite dans le cadre de la Théorie des Graphes mais n'a semble-t-il pas trouvé, pour le moment,

3. Nous noterons $E_{p(i)}$ le cluster parent du cluster E_i et nous supposerons que $E_i \cap E_{p(i)} = \emptyset$ si E_i est le cluster racine.

4. Pour tout $Y \subseteq X$, $G[Y] = (Y, C_Y)$ est le sous-graphe de $G = (X, C)$ induit par Y où $C_Y = \{\{x, y\} \in C | x, y \in Y\}$.

d'écho au sein de cette communauté. Plus précisément, la notion de *Connected Tree-Width* est présentée dans [18] [5] et a fait l'objet, lors de sa définition, d'une étude portant sur ses propriétés combinatoires. Les questions algorithmiques, comme notamment le problème de son calcul en termes de complexité ou bien la proposition d'algorithme l'approximant, n'ont pas été évoquées. Cette contribution fournit en fait un théorème central indiquant une majoration de ce nouveau paramètre en fonction de la largeur arborescente et de la longueur maximum de ses cycles géodésiques (cycles pour lesquels la distance entre toute paire de sommets du cycle est égale à la longueur du plus court chemin empruntant le cycle). Du point de vue terminologique, il nous apparaît préférable d'utiliser les termes Bag-Connected Tree-Decomposition et Bag-Connected Tree-Width qui n'ont à ce jour, et à notre connaissance, pas encore fait l'objet d'une autre définition. Nous présentons donc la notion de Bag-Connected Tree-Decomposition, qui correspond aux décompositions arborescentes telles que chaque cluster E_i est connexe (i.e. $G[E_i]$ est un graphe connexe).

Définition 2 *Étant donné un graphe $G = (X, C)$, une* Bag-Connected Tree-Decomposition *de G est une décomposition arborescente (E, T) de G telle que pour tout $E_i \in E$, le sous-graphe $G[E_i]$ de G induit par E_i est un graphe connexe. La largeur d'une Bag-Connected Tree-Decomposition (E, T) est égale à $max_{i \in I}|E_i| - 1$ et la* Bag-Connected Tree-Width w_c *est la largeur minimale pour toutes les bag-connected tree-decompositions de G.*

Étant donné un graphe $G = (X, C)$ de largeur arborescente w, on a nécessairement $w \leq w_c$. Néanmoins, si G est un graphe triangulé, $w = w_c$. Sinon, par exemple pour les graphes formés d'un cycle de longueur n et sans corde, la Bag-Connected Tree-Width vaut $\lceil \frac{n}{2} \rceil$.

Une question naturelle maintenant est liée au calcul des Bag-Connected Tree-Decompositions optimales, c'est-à-dire de largeur égale à w_c. Nous montrons que ce problème, comme pour le cas des décompositions arborescentes, est NP-difficile (la preuve est donnée en annexe).

Théorème 1 *Calculer une Bag-Connected Tree-Decomposition optimale est NP-difficile.*

Nous avons vu que pour la résolution de CSP, il n'est pas nécessaire de trouver une décomposition arborescente optimale, et c'est même souvent souhaitable. Aussi, nous proposons ici un algorithme appelé *Bag-Connected-TD*, qui calcule, pour un graphe

5. À ne pas confondre avec la notion du même nom introduite dans [9] mais qui est différente.

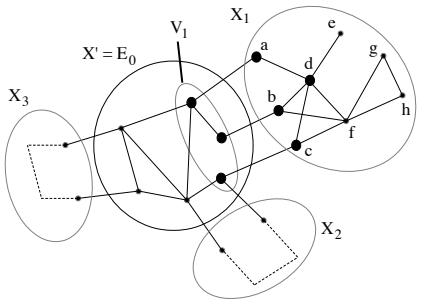


FIGURE 3 – Illustration de *Bag-Connected-TD*

$G = (X, C)$, une Bag-Connected Tree-Decomposition en temps polynomial, bien sûr, sans aucune garantie quant à son optimalité

La première étape de l'algorithme calcule un premier cluster, noté E_0, constitué d'un sous-ensemble de sommets connexes. X' notera par la suite l'ensemble des sommets déjà traités. Cet ensemble est initialisé à E_0. Cette première étape peut se faire facilement, en utilisant une méthode heuristique. Dans la suite, nous noterons $X_1, X_2, \ldots X_k$ les composantes connexes du sous-graphe $G[X \backslash E_0]$ induit par la suppression dans G des sommets de E_0. Chacun de ces ensembles X_i est inséré dans une file F. Pour chaque élément X_i supprimé de la file F, on notera $V_i \subseteq X$ l'ensemble des sommets de X' qui sont adjacents à au moins un sommet de X_i. On peut noter que V_i (qui peut être connexe ou non) est un séparateur du graphe G puisque la suppression de V_i dans G rend G non connexe (X_i étant déconnecté du reste de G). Un nouveau cluster E_i est alors initialisé par cet ensemble V_i. Nous considérons alors le sous-graphe de G induit par V_i et X_i, c'est-à-dire $G[V_i \cup X_i]$. Nous choisissons un premier sommet $x \in X_i$ qui est connecté à au moins un sommet de E_i (donc un sommet de V_i). Ce sommet est ajouté à E_i. Si $G[E_i]$ est connexe, le processus est arrêté puisque nous sommes sûrs que E_i sera un nouveau cluster connexe. Sinon, on continue, en prenant un autre sommet de X_i.

La figure 3 présente le calcul de E_1, le second cluster (après E_0), lors du premier passage dans la boucle. Après l'ajout des sommets a, b et c, le sous-graphe $G[V_1 \cup \{a, b, c\}]$ n'est pas connexe. Si le prochain sommet atteint est d, il est rajouté à E_1, et donc $E_1 = V_1 \cup \{a, b, c, d\}$ constitue un nouveau cluster connexe, stoppant ainsi la recherche dans $G[V_1 \cup X_1]$.

Quand ce calcul est terminé, nous rajoutons les sommets de E_i à X' et nous calculons $X_{i_1}, \ldots X_{i_{k_i}}$ les composantes connexes du sous-graphe $G[X_i \backslash E_i]$. Chacune est alors rajoutée à la file F. Dans l'exemple de la figure 3, deux composantes connexes seront calculées, $\{e\}$ et $\{f, g, h\}$. Le processus se poursuit tant que la file n'est pas vide. Dans l'exemple, et pour la partie droite du graphe, l'algorithme calculera 3 clusters connexes :

Algorithme 1: *Bag-Connected-TD*

Input : Un graphe $G = (X, C)$
Output : Un ensemble de clusters $E_0, \dots E_m$ d'une bag-connected tree-decomposition de G

1 Choisir un premier cluster connexe E_0 in G
2 $X' \leftarrow E_0$
3 Soient $X_1, \dots X_k$ les composantes connexes de $G[X \backslash E_0]$
4 $F \leftarrow \{X_1, \dots X_k\}$
5 **while** $F \neq \emptyset$ **do** `/* calcule un nouveau cluster` E_i `*/`
6 Enlever X_i de F
7 Soit $V_i \subseteq X'$ le voisinage de X_i dans G
8 $E_i \leftarrow V_i$
9 Recherche dans $G[V_i \cup X_i]$ à partir de V_i plus $x \in X_i$. Chaque fois qu'un nouveau sommet x est atteint, il est ajouté à E_i. Le processus s'arrête dès que le sous-graphe $G[E_i]$ est connexe
10 **if** V_i *appartient aux clusters déjà trouvés* **then**
11 Détruire le cluster V_i (parce que $V_i \subsetneq E_i$)
12 $X' \leftarrow X' \cup E_i$
13 Soient $X_{i_1}, X_{i_2}, \dots X_{i_{k_i}}$ les composantes connexes de $G[X_i \backslash E_i]$
14 $F \leftarrow F \cup \{X_{i_1}, X_{i_2}, \dots X_{i_{k_i}}\}$

$\{d, e\}$, $\{b, c, d, f\}$ et $\{f, g, h\}$.

Notons que la ligne 11 est utile uniquement quand l'algorithme construit un cluster E_i contenant tous les sommets d'un cluster déjà obtenu E_j (i.e. $E_j \subsetneq E_i$). Dans un tel cas, le cluster E_j peut être supprimé car il sera inutile dans la décomposition arborescente.

Nous établissons maintenant la validité de l'algorithme, puis sa complexité temporelle (les preuves sont données en annexe).

Théorème 2 *Bag-Connected-TD calcule les clusters d'une Bag-Connected Tree-Decomposition.*

Théorème 3 *La complexité en temps de l'algorithme Bag-Connected-TD est* $O(n(n+e))$.

D'un point de vue pratique, on peut supposer que le choix du premier cluster E_0 peut être crucial pour la qualité de la décomposition qui est en cours de calcul. De même, le choix de sommet x, sélectionné dans la ligne 9 peut être d'une importance considérable. Pour ces deux choix, des heuristiques peuvent bien sûr être utilisées. Cette question est abordée dans la partie suivante. Cependant, un choix particulier de ces heuristiques permet, sans aucune modification de la complexité, de calculer des décompositions arborescentes optimales pour le cas des graphes triangulés. Supposons que le premier cluster E_0 soit une clique maximale. Cela peut être réalisé efficacement en utilisant une approche gloutonne. Maintenant, pour le choix du sommet x à la ligne 9, nous considérons le sommet qui possède le nombre maximum de voisins dans l'ensemble V_i. Comme dans un graphe triangulé, tous les clusters d'une décomposition arborescente optimale sont des cliques, nécessairement, V_i étant une clique, x sera relié à tous les sommets de V_i et donc, E_i sera une clique. Progressivement, chaque clique maximale sera trouvée et la décomposition arborescente sera optimale. Les lignes 10-11 seront utilisées pour le cas de cliques maximales incluant plus d'un sommet x issu d'une nouvelle composante connexe. Dans tous les cas, l'intérêt pratique de ce type de décomposition est basé à la fois sur l'efficacité de son calcul, mais aussi sur l'apport qu'elle peut avoir pour la résolution de CSP. Cette question est abordée dans la partie suivante.

5 Résultats expérimentaux

Dans cette section, nous comparons l'efficacité de la résolution ainsi que les valeurs des paramètres structurels, selon que l'on utilise des décompositions arborescentes produites par Min-Fill ou bien par l'algorithme *Bag-Connected-TD*. En ce qui concerne le choix du premier cluster dans les calculs de Bag-Connected Tree-Decompositions, nous calculerons de façon gloutonne une clique maximale du réseau de contraintes [6]. Dans l'algorithme *Bag-Connected-TD*, pour choisir le prochain sommet, nous avons considéré six heuristiques dont nous ne présentons ici que les 4 meilleures :

- NV1 : le prochain sommet est un sommet situé dans le voisinage des sommets précédemment choisis.
- NV2 : les sommets sont traités dans l'ordre décroissant des degrés.
- NV3 : les sommets sont traités selon l'ordre dans lequel ils sont visités par un parcours *en largeur d'abord* du graphe à partir des sommets de V_i.
- NV4 : nous choisissons comme prochain sommet celui qui possède le nombre maximum de voisins dans l'ensemble V_i.

La résolution de CSP est réalisée par BTD basée sur MAC, en utilisant l'heuristique de choix de variables *dom/wdeg* [3]. Nous choisissons comme cluster racine celui qui maximise le rapport $\frac{e}{n-1}$. Ce choix donne de meilleurs résultats que ceux introduits dans [15]. Les temps d'exécution de la décomposition pour Min-Fill et *Bag-Connected-TD* sont similaires et sont inclus dans ceux de BTD et ils se révèlent relativement négligeables au regard du temps de résolution. Toutes les implémentations sont écrites en C++. Elles ont été réalisées sur un PC sous Linux doté d'un processeur Intel Pentium IV 3, 2 GHz avec 1 Go de mémoire.

5.1 Instances pour lesquelles Min-Fill produit des clusters non connexes

Dans ce paragraphe, nous comparons, du point de vue de l'efficacité de la résolution, les bag-connected tree-decompositions à celles qui contiennent des clusters non connexes. Pour ce faire, nous considérons

6. Rappelons-nous que nous utilisons les 2-sections pour les instances non binaires.

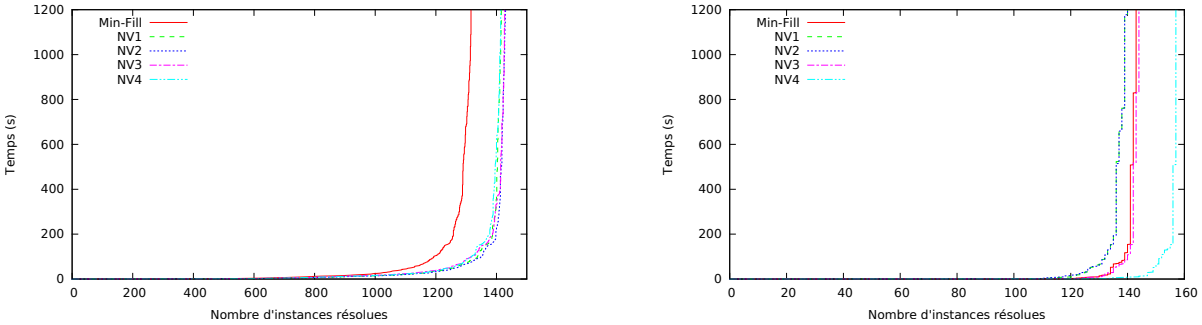


FIGURE 4 – Nombre d'instances résolues pour chaque décomposition arborescente considérée pour les benchmarks pour lesquels Min-Fill produit des clusters non connexes (a) et pour les instances pour lesquelles Min-Fill produit une décomposition arborescente avec des clusters connexes (b).

1 597 instances (d'arité quelconque) parmi les 2 310 instances de la compétition CSP 2008 et pour lesquelles Min-Fill produit une décomposition arborescente possédant au moins un cluster non connexe. Sont exclues des résultats, les instances qui n'ont pu être résolues sans dépasser la limite de temps (à savoir 1 200 secondes) ou qui contiennent des contraintes globales (car elles ne sont pas encore mises en œuvre dans notre solveur). Parmi les instances étudiées, nous pouvons notamment trouver des instances issues des familles *rlfap*, *fapp*, *modifiedRenault*, *graphColoring*, *bqwh* ou *travellingSalesman*.

La figure 4 (a) présente le nombre cumulé d'instances résolues pour chaque type de décomposition arborescente. Tout d'abord, nous pouvons observer qu'en utilisant chacune des bag-connected tree-decompositions, BTD résout plus d'instances qu'en utilisant les décompositions arborescentes déconnectées produites par Min-Fill. On notera que cette observation reste vraie si nous utilisons les décompositions produites par les heuristiques non présentées. Le meilleur nombre d'instances résolues est obtenu grâce aux décompositions produites par les heuristiques NV2 et NV3. Ces décompositions permettent de résoudre respectivement 114 et 111 instances de plus qu'avec Min-Fill. Celles à base de NV1 et NV4 sont proches l'une de l'autre. Par ailleurs, pour toute les décompositions, la plupart des instances sont résolues en moins de 60 secondes.

Afin de comparer plus équitablement les temps d'exécution, nous ne considérons maintenant que les instances qui sont résolues par BTD pour toutes les décompositions considérées, y compris Min-Fill. Le temps d'exécution pour résoudre les 1 230 instances en utilisant les décompositions basées sur Min-Fill est de 50 669 secondes alors qu'en utilisant les décompositions connexes basées sur des NV1, cela ne nécessite que 32 372 secondes. Les décompositions basées sur NV2 sont relativement proches de NV1, à savoir 33 202 secondes. Celles qui sont basées sur NV3 et NV4 sont légèrement plus lentes avec respectivement 36 420 et 36 087 secondes. On notera que les deux autres heuristiques (non présentées ici) surpassent également la décomposition Min-Fill.

Si nous nous concentrons sur les 329 instances ayant une structure bien adaptée à la décomposition (i.e. de largeur relativement faible par rapport au nombre de variables), de nouveau, on observe la même tendance, à savoir que BTD exploité sur des bag-connected tree-decompositions est plus performant que BTD avec Min-Fill. Le meilleur temps d'exécution est obtenu par BTD en utilisant NV1 avec 5 698 secondes, tandis que le pire l'est en utilisant Min-Fill avec 13 641 secondes. De plus, BTD utilisant NV2 et NV4 fournit des résultats voisins l'un de l'autre avec respectivement 6 137 et 6 010 secondes tandis que BTD avec NV3 nécessite 8 483 secondes.

Enfin, si l'on compare ces résultats avec ceux obtenus par un algorithme énumératif classique comme MAC, nous pouvons noter que certaines instances résolues par BTD avec certains NVi ne sont pas résolues par MAC et inversement. Nous observons également que MAC se comporte parfois mieux, parfois moins bien que BTD basé sur des décompositions connexes. Toutefois on peut estimer que globalement, les résultats sont similaires. Ceci s'explique par le fait que la plupart des 1 597 instances que nous considérons sont loin d'avoir une structure adaptée à la résolution par décomposition. En revanche, lorsque la structure possède des caractéristiques intéressantes, BTD surclasse clairement MAC. Par exemple, BTD exploitant une décomposition basée sur NV3 ne nécessite que 856 secondes pour résoudre 10 instances sur les 12 de la famille *rlfapScens11* tandis que MAC n'en résout que 8 en 1 595 secondes. Par ailleurs, pour résoudre ces huit instances, BTD ne nécessite que 63 secondes, ce qui correspond à un comportement 25 fois plus rapide que celui de MAC.

5.2 Instances pour lesquelles Min-Fill produit des décompositions arborescentes connexes

Nous abordons ici brièvement le comportement de BTD pour résoudre les instances pour lesquelles Min-Fill produit des bag-connected tree-decompositions. Bien sûr, pour ces instances, Min-Fill et l'algorithme *Bag-Connected-TD* ne produisent pas nécessairement les mêmes décompositions. Nous nous concentrons ici sur les cas les plus pertinents, c'est-à-dire les 191 instances ayant une structure adaptée à une approche de résolution par décomposition.

Comme le montre la figure 4 (b), l'utilisation de BTD basé sur Min-Fill permet de résoudre plus d'instances que BTD basé sur NV1 ou NV2 (143 contre 140 instances), mais un nombre inférieur à BTD basé sur NV3 ou NV4 qui en résolvent respectivement 144 et 157. Si nous concentrons notre étude sur les 132 instances qui sont résolues par BTD pour toutes les décompositions arborescentes considérées, y compris avec Min-Fill, BTD basé sur NV3, NV4 ou Min-Fill conduit aux meilleurs cumuls de temps d'exécution avec respectivement 1 283, 1 298 et 1280 secondes tandis que BTD basé sur NV1 ou NV2 est bien plus lent, avec respectivement 2 226 et 2 265 secondes.

5.3 Comparaison des paramètres structurels

Le tableau 1 présente la valeur des paramètres structurels pour certaines instances qui sont représentatives des tendances générales. Sans surprise, Min-Fill produit des décompositions de largeur plus petite, mais avec un plus grand nombre de clusters, que celles produites par *Bag-Connected-TD*. Cependant, si dans certains cas, la largeur obtenue par *Bag-Connected-TD* est significativement plus grande que celle fournie par Min-Fill (e.g. la largeur produite par NV3 pour l'instance *squares-23-23*), dans d'autres cas, elle reste relativement proche, voire parfois même égale à celle obtenue par Min-Fill. Cela se produit notamment pour l'instance *renault-mod-33_ext* mais aussi pour les instances pour lesquelles Min-Fill produit une bag-connected tree-decomposition (cf. partie (b) du tableau 1). Nous observons également que la qualité de la largeur obtenue grâce à *Bag-Connected-TD* peut varier considérablement selon les instances. Si NV1 présente souvent la meilleure largeur parmi celles calculées par l'algorithme *Bag-Connected-TD*, celle-ci est cependant parfois inférieure pour NV3 ou NV4 (e.g. pour l'instance *mps-red-qnet1*).

Enfin, pour ce qui concerne le paramètre s, les tendances constatées se révèlent similaires à celle observées pour la largeur.

6 Conclusion

Dans cet article, nous avons introduit, dans le cadre de la résolution de CSP, le concept de Bag-Connected Tree-Decomposition. Après avoir montré l'intérêt de cette notion et proposé un premier algorithme polynomial qui calcule de telles décompositions, nous avons démontré expérimentalement la pertinence de cette approche car elle permet d'améliorer considérablement la résolution de CSP basée sur les méthodes de décomposition. En effet, ce type de décomposition permet de résoudre beaucoup plus d'instances et d'améliorer le temps d'exécution, par exemple, d'environ 63% dans le cas des instances pour lesquelles Min-Fill produit des clusters non connexes. En outre, ces décompositions peuvent aussi, pour les benchmarks bien structurés, rendre BTD bien plus performant que MAC, en étant en particulier jusqu'à 25 fois plus rapide (par exemple pour la famille d'instances *rlfapScens11*).

La première extension à ce travail porte sur l'étude de ces décompositions dans le domaine plus général des modèles graphiques en IA. Il s'agit d'exploiter cette notion avec d'autres classes de méthodes telles que l'Hypertree-Decompositions, And/Or Search, ou encore Bucket Elimination. Cette démarche est en effet particulièrement justifiée par le fait que, même si certaines de ces approches sont en théorie basées sur d'autres paramètres (par exemple l'Hypertree-Width), les implémentations efficaces connues s'appuient généralement sur des algorithmes de calcul de décompositions arborescentes (par exemple Min-Fill pour l'Hypertree-Decomposition [8]). Un autre développement qui nous semble prometteur porte sur l'utilisation des Bag-Connected Tree-Decompositions dans le domaine de l'optimisation ou encore des problèmes de comptage de solutions.

Enfin, il serait aussi judicieux de développer le travail balbutiant relatif à une étude théorique de ce nouveau paramètre, d'un point de vue mathématique, et portant par exemple sur les propriétés fondamentales de la Bag-Connected Tree-Width. Le travail préliminaire présenté dans [18] offre un premier élément de réponse mais limité pour le moment à une seule borne supérieure de ce paramètre. Une autre question pourrait porter sur la mise en évidence de classes de graphes pour lesquelles ce paramètre est facile à calculer (i.e. de complexité polynomiale), ou bien alors proche de la largeur arborescente. Ou bien encore, de déterminer des problèmes qui sont difficiles quand la largeur arborescente est bornée par une constante, et qui deviendraient faciles quand la Bag-Connected Tree-Width est bornée par une constante.

TABLE 1 – Valeur des paramètres structurels pour certaines instances pour lesquelles Min-Fill produit des clusters non connexes (a), et pour lesquels Min-Fill produit une bag-connected tree-decomposition (b).

	Instances	n	e	Min-Fill		NV1		NV2		NV3		NV4	
				w^+	s	w_c^+	s	w_c^+	s	w_c^+	s	w_c^+	s
(a)	2-insertions-4-3	149	541	38	34	66	54	95	14	101	66	58	57
	ewddr2-10-by-5-9	50	265	16	15	22	17	21	20	26	23	45	37
	renault-mod-33_ext	111	133	11	11	12	11	14	11	17	15	16	13
	scen7	400	2 865	33	29	90	48	319	9	116	94	81	34
	squares-23-23	1 058	1 268	45	4	45	5	45	5	235	88	45	26
	fapp06-0500-1	500	3 478	221	210	286	284	286	284	314	314	313	248
	js-taillard-15-100-4	225	1 785	86	70	114	102	121	97	129	102	210	197
(b)	mps-red-qnet1	5 380	621	970	773	1 272	1 265	1 272	1 265	978	954	998	974
	anna-9	138	493	12	12	14	14	14	14	16	15	14	13
	haystacks-10	100	459	9	1	9	1	9	1	9	1	9	1
	renault-mod-8_ext	111	126	11	11	11	11	12	11	13	12	11	11
	qwh-15-106-9_ext	225	2 324	99	99	102	102	102	102	103	103	173	168

Références

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

[2] C. Berge. *Graphs and Hypergraphs*. Elsevier, 1973.

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

[4] C. Cabon, S. de Givry, L. Lobjois, T. Schiex, and J. P. Warners. Radio Link Frequency Assignment. *Constraints*, 4 :79–89, 1999.

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

[6] R. Dechter and Y. El Fattah. Topological Parameters for Time-Space Tradeoff. *Artificial Intelligence*, 125 :93–118, 2001.

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

[8] A. Dermaku, T. Ganzow, G. Gottlob, B. J. McMahan, N. Musliu, and M. Samer. Heuristic methods for hypertree decomposition. In *MICAI*, pages 1–11, 2008.

[9] P. Fraigniaud and N. Nisse. Connected treewidth and connected graph searching. In *LATIN*, pages 479–490, 2006.

[10] E. Freuder. A Sufficient Condition for Backtrack-Free Search. *JACM*, 29 (1) :24–32, 1982.

[11] M. Golumbic. *Algorithmic Graph Theory and Perfect Graphs*. Academic Press, New York, 1980.

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

[13] M. Gyssens, P. Jeavons, and D. Cohen. Decomposing constraint satisfaction problems using database techniques. *Artificial Intelligence*, 66 :57–89, 1994.

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

[15] P. Jégou, S. N. Ndiaye, and C. Terrioux. An extension of complexity bounds and dynamic heuristics for tree-decompositions of CSP. In *Proceedings of CP*, pages 741–745, 2006.

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

[17] U. Kjaerulff. Triangulation of Graphs - Algorithms Giving Small Total State Space. Technical report, Judex R.R. Aalborg., Denmark, 1990.

[18] Malte Muller. Connected tree-width. *CoRR*, abs/1211.7353, 2014.

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

[20] D. J. Rose. A graph theoretic study of the numerical solution of sparse positive denite systems of linear equations. In *Graph Theory and Computing*, pages 183–217. Academic Press, 1972.

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

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

7 Annexes

Preuve du Théorème 1 : Nous proposons une réduction polynomiale du problème du calcul d'une décomposition arborescente optimale vers ce problème. Considérons un graphe $G = (X, C)$ de largeur arborescente w, la décomposition arborescente de G associée étant (E, T). Considérons le graphe G' obtenu en ajoutant à G un sommet universel x, i.e. un sommet qui est relié à tous les sommets de G. Notons qu'à partir de (E, T), nous pouvons obtenir une décomposition arborescente pour G' en ajoutant dans chaque cluster $E_i \in E$, le sommet x. Il s'agit d'une Bag-Connected Tree-Decomposition puisque chaque cluster est nécessairement connexe (au moins par des chemins contenant x) et sa largeur vaut $w + 1$. Pour montrer que cet ajout de sommet définit une réduction, il suffit de montrer que w est la largeur d'arbre de G si et seulement si la Bag-Connected Tree-Width w_c de G' est $w + 1$.

1. ($\Rightarrow$) Nous savons qu'au plus, la largeur de la décomposition arborescente considérée de G' est $w + 1$ puisque les clusters de cette décomposition arborescente sont connexes et que sa largeur est $w+1$. Supposons que $w_c \leq w$, et que donc, il existe une Bag-Connected Tree-Decomposition de G' de largeur au plus w. En utilisant la décomposition arborescente de G', on peut définir le même arbre, mais en supprimant le sommet x pour obtenir une décomposition arborescente de G de largeur $w-1$, ce qui contredit l'hypothèse.
2. ($\Leftarrow$) Avec le même type d'arguments que ci-dessus, nous savons que la largeur arborescente w de G est au plus $w_c - 1$. Et par construction, elle ne peut pas être strictement inférieure à $w_c - 1$. Donc, c'est exactement $w_c - 1$.

De plus, la construction de G' est possible en temps linéaire. □

Preuve du Théorème 2 : Il suffit de prouver les lignes 5-14 de l'algorithme. Nous montrons d'abord que l'algorithme s'arrête. À chaque passage dans la boucle, au moins un sommet sera rajouté à l'ensemble X' et ce sommet n'apparaîtra pas plus tard dans un nouvel élément de la file d'attente puisque ces éléments sont définis par les composantes connexes de $G[X_i \backslash E_i]$, un sous-graphe qui contient strictement moins de sommets qu'il n'y en a dans X_i. Ainsi, après un nombre fini d'étapes, l'ensemble $X_i \backslash E_i$ sera un ensemble vide, et donc aucun nouvel ajout à F ne sera possible.

Nous montrons maintenant que l'ensemble des clusters $E_0, E_1, \ldots E_m$ induit une bag-connected tree-decomposition. Par construction, chaque nouveau cluster est connexe. Donc, nous devons juste prouver qu'ils induisent une décomposition arborescente. Nous le prouvons par induction sur les clusters ajoutés, en montrant que tous ces clusters ajoutés vont induire une décomposition arborescente du graphe $G[X']$.

Initialement, le premier cluster E_0 induit une décomposition arborescente du graphe $G[E_0] = G[X']$.

Pour l'induction, notre hypothèse est que l'ensemble des clusters déjà ajoutés $E_0, E_1, \ldots E_{i-1}$ induit une décomposition arborescente du graphe $G[E_0 \cup E_1 \cup \ldots \cup E_{i-1}]$. Considérons maintenant l'ajout de E_i. Nous montrons que par construction, $E_0, E_1, \ldots E_{i-1}$ et E_i induit une décomposition arborescente du graphe $G[X']$ en montrant que les trois conditions (i), (ii) et (iii) de la définition des décompositions arborescentes sont satisfaites.

(i) Chaque nouveau sommet ajouté dans X' appartient à E_i

(ii) Chaque nouvelle arête dans $G[X']$ est à l'intérieur du cluster E_i.

(iii) On peut considérer deux cas différents pour un sommet $x \in E_i$, sachant que pour les autres sommets, la propriété est déjà satisfaite par l'hypothèse d'induction :

  (a) $x \in E_i \backslash V_i$: dans ce cas, x n'apparaît pas dans un autre cluster que E_i et donc, la propriété est vérifiée.

  (b) $x \in V_i$: dans ce cas, par hypothèse d'induction, la propriété a déjà été vérifiée.

Finalement, il est facile de voir que si la ligne 11 est appliquée, on obtient bien les clusters d'une décomposition arborescente du graphe $G[X']$. □

Preuve du Théorème 3 : Les lignes 1-4 sont réalisables en temps linéaire, soit $O(n + e)$, puisque le coût de calcul des composantes connexes de $G[X \backslash E_0]$ est borné par $O(n + e)$. Néanmoins, nous pouvons noter que la ligne 1 peut être réalisée par une heuristique éventuellement plus coûteuse, de sorte à obtenir un premier cluster plus pertinent, mais en se limitant à en $O(n(n+e))$ afin de ne pas accroître la complexité globale de l'algorithme. Nous analysons maintenant le coût de la boucle (ligne 5). Tout d'abord, notons qu'il y a nécessairement moins de n insertions dans la file d'attente F car à chaque passage dans la boucle, on est assuré qu'au moins, un nouveau sommet aura été rajouté dans X', et donc supprimé de l'ensemble des sommets n'ayant pas encore été traités. Nous analysons maintenant le coût de chaque traitement associé à l'ajout d'un nouveau cluster, dont nous donnons pour chacun, sa complexité globale.

– Ligne 6 : l'obtention du premier élément X_i de F est bornée par $O(n)$, et donc globalement par $O(n^2)$.

- Ligne 7 : l'obtention du voisinage $V_i \subseteq X'$ de X_i dans G est bornée par $O(n+e)$, et donc globalement par $O(n(n+e))$.
- Ligne 8 : cette étape est réalisable en $O(n)$, soit globalement en $O(n^2)$.
- Ligne 9 : le coût de la recherche dans $G[V_i \cup X_i]$ débutant avec les sommets de V_i et $x \in X_i$ est borné par $O(n+e)$. Puisque la boucle *while* s'exécute au plus n fois, le coût global de la recherche dans ces sous-graphes est borné par $O(n(n+e))$. En outre, pour chaque nouveau sommet x ajouté, la connexité de $G[E_i]$ est testée avec un coût supplémentaire borné par $O(n+e)$. On peut remarquer qu'un tel sommet n'est ajouté au plus qu'une fois, donc globalement, le coût de ce test est borné par $O(n(n+e))$. Donc, le coût de la ligne 9 est globalement borné par $O(n(n+e))$.
- Lignes 10-11 : en utilisant une structure de données adaptée, cette étape peut être réalisée en $O(n)$, soit globalement en $O(n^2)$.
- Ligne 12 : cette étape est réalisable en $O(n)$, soit globalement en $O(n^2)$.
- Ligne 13 : le coût de la recherche des composantes connexes de $G[X_i \backslash E_i]$ est borné par $O(n+e)$. Ainsi, globalement, le coût de cette étape est $O(n(n+e))$.
- Ligne 14 : l'insertion de X_{i_j} dans F est réalisable en $O(n)$, soit globalement en $O(n^2)$ puisqu'il y a moins de n insertions dans F.

Finalement, la complexité temporelle de l'algorithme *Bag-Connected-TD* est $O(n(n+e))$. □

Recherche Multiobjectif à Voisinage Large

Pierre Schaus **Renaud Hartert**

UCLouvain, ICTEAM
Place saint barbe 2
1348 Louvain-la-Neuve, Belgium
{pierre.schaus, renaud.hartert}@uclouvain.be

Résumé

La Recherche à Voisinage Large (LNS) est un framework qui permet de combiner l'expressivité de la programmation par contraintes avec l'efficacité de la recherche locale pour résoudre les problèmes d'optimisation combinatoire. Ce papier introduit une extension de LNS pour résoudre les problèmes combinatoires avec plus d'un objectif à optimiser simultanément. Ce nouveau framework, nommé MO-LNS, maintient non pas la meilleure solution courante mais une population de meilleures solutions courantes. A chaque itération, une de ces solutions est sélectionnée, relâchée et optimisée dans le but de strictement améliorer la qualité de la population. L'efficacité de cette approche est évaluée sur plusieurs exemples.

Abstract

Large neighborhood search (LNS) is a framework that combines the expressiveness of constraint programming with the efficiency of local search to solve combinatorial optimization problems. This paper introduces an extension of LNS, called Multi-Objective LNS (MO-LNS), to solve multi-objective combinatorial optimization problems ubiquitous in practice. The idea of MO-LNS is to maintain a set of nondominated solutions rather than just one best-so-far solution. At each iteration, one of these solutions is selected, relaxed and optimized in order to strictly improve the hypervolume of the maintained set of nondominated solutions. We show experimentally the efficiency of this approach on two multi-objective combinatorial optimization problems.

1 Introduction

Les problèmes d'optimisation multiobjectif sont omniprésents dans le monde réel. En effet, il n'est pas rare d'être confronté à plusieurs objectifs à optimiser simultanément (e.g. coût et qualité). Dans le cas où aucune préférence ne peut être établie a priori, il est souvent nécessaire de proposer à l'utilisateur un ensemble de solutions contenant tous les compromis optimaux entre ces objectifs. L'utilisateur, ensuite, sélectionne lui-même la solution qui correspont le plus à ses besoins.

Il n'est pas étonnant que la théorie et la méthodologie de l'optimisation multiobjectif fassent l'objet d'un intérêt grandissant [10]. Au moment d'écrire cet article, les meta-heuristiques hybrides entre recherche locale et algorithmes évolutionnaires obtiennent les meilleurs résultats sur de nombreux problèmes multiobjectif standards i.e. le problème du voyageur de commerce multiobjectif [8]. Malheureusement, et en dépit des fonctionnalités proposées par des bibliothèques telles que ParadisEO [1] et jMetal [2], ces approches sont encore loin d'être déclaratives. En effet, l'utilisateur doit toujours fournir de nombreuse fonctions de bas niveau (e.g. génération du voisinage, mutation) qui nécessitent une importante connaissance du problème à résoudre [4].

La programmation par contraintes est une technique d'optimisation reconnue pour son aspect déclaratif tout en étant compétitive pour résoudre les problèmes d'optimisation combinatoire à un seul objectif [13, 15, 20]. En particulier, le framework de recherche à voisinage large (LNS), combinant l'efficacité de la recherche locale avec l'expressivité de la programmation par contraintes, fut appliqué avec succès à de nombreux problèmes industriels complexes et de grandes tailles [9, 12, 14, 19]. Dans ce travail, nous présentons une simple extension de la recherche à voisinage large pour résoudre les problèmes d'optimisation cominatoire multiobjectif. L'efficacité et la flexibilité de ce nouveau framework, nommé recherche multiobjectif à voisinage large (MO-LNS), est vérifiée sur plusieurs instances du problème du sac à dos [22] et du

problème d'allocation de conteneurs [18].

Ce document est structuré comme suit...

2 Prérequis

Un problème d'optimisation multiobjectif est un quadruple $\mathcal{M} = \langle \mathbf{X}, \mathbf{D}, \mathbf{C}, \mathbf{F} \rangle$ où $\mathbf{X} = \{x_1, \ldots, x_n\}$ est un ensemble de variables, $\mathbf{D} = \{D_1, \ldots, D_n\}$ est l'ensemble des domaines de ces variables, $\mathbf{C}$ est un ensemble de contraintes sur les variables et $\mathbf{F} = \{f_1, \ldots, f_m\}$ est un ensemble de fonctions objectif à minimiser simultanément. Dans la suite de ce document, nous supposons que m variables $obj_1, \ldots, obj_m$ (que nous appelons variables objectif) ont été ajoutées à l'ensemble de variables $\mathbf{X}$ et contraintes à être égale à la valeur prise par leur fonction objectif i.e. $obj_i = f_i(\mathbf{X})$. En particulier, la valeur minimale et la valeur maximale du domaine d'une variable objectif obj_i correspondent aux bornes de cet fonctions objectif par rapport à un assignment partiel des variables de $\mathbf{X}$.

Une solution d'un problème multiobjectif $\mathcal{M}$ est une assignation complète des variables de $\mathbf{X}$ qui satisfait toutes les contraintes de l'ensemble $\mathbf{C}$. Dans ce travail, nous représentons une solution sol par sa projection dans l'espace objectif $(sol_1, \ldots, sol_m)$ où sol_i est la valeur assignée à la variable objectif obj_i.

Comme il est peu commun qu'une solution soit à la fois optimale pour tous les objectifs en même temps, nous cherchons à définir un ordre partiel entre le vecteur objectif des différentes solutions. Parmis ces ordonnancements, la dominance faible de Pareto (que nous appelons dominance dans le reste de ce document) est un choix commun (voir FIGURE 1).

Définition 1 (Dominance faible). *Considérons $sol = (sol_1, \ldots, sol_m)$ et $sol' = (sol'_1, \ldots, sol'_m)$ deux solutions d'un problème multiobjectif. Nous disons que sol domine sol', dénoté $sol \preceq sol'$, si et seulement si :*

$$\forall i \in \{1, \ldots, m\} : sol_i \leq sol'_i \tag{1}$$

Soit $sols(\mathcal{M})$ l'ensemble de toutes les solutions d'un problème multiobjectif $\mathcal{M}$. Une solution $sol \in sols(\mathcal{M})$ est dite *efficace* si et seulement si il n'existe pas de solution $sol' \in sols(\mathcal{M})$ qui domine sol :

$$\nexists sol' \in sols(\mathcal{M}) : sol' \neq sol \wedge sol' \preceq sol. \tag{2}$$

En d'autres mots, une solution est dite efficace si il n'est pas possible d'améliorer la valeur d'un objectif sans dégrader la valeur d'au moins un autre objectif (i.e. une solution efficace est un compromis optimal entre les objectifs).

Résoudre un problème multiobjectif revient souvent à trouver l'ensemble de toutes les solutions efficaces i.e. *l'ensemble efficace* ou *front de Pareto* (voir FIGURE 2 illustre le front de Pareto d'un ensemble de solutions arbitraire). Malheureusement, la taille du front de Pareto croit souvent de façon exponentielle avec la taille du problème à résoudre [3]. De ce fait, en pratique, seule une approximation de cet ensemble peut être trouvée en des temps et quantités de mémoire raisonnables. Nous appelons une telle approximation une archive.

Définition 2 (Archive). *Une archive $\mathcal{A}$ est un ensemble de solutions sans dominance i.e. un ensemble de solutions tel qu'aucune solution de $\mathcal{A}$ ne domine une autre solution de $\mathcal{A}$:*

$$\forall sol \in \mathcal{A}, \nexists sol' \in \mathcal{A} : sol' \neq sol \wedge sol' \preceq sol. \tag{3}$$

Clairement, le front de Pareto est une archive.

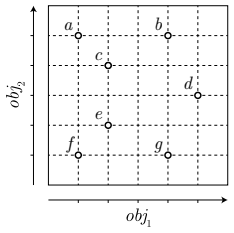


FIGURE 1 – La solution e domine les solutions b, c et d tout en étant dominée par f. Les solutions a et g ne dominent pas e et ne sont pas dominées par e.

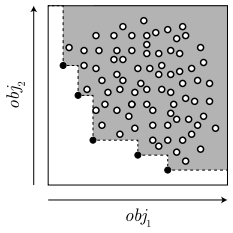


FIGURE 2 – Les solutions efficaces de cet espace biobjectif de solutions sont coloriées en noir. Le front de Pareto est l'ensemble de toutes les solutions efficaces.

Une archive peut servir de base pour diviser l'espace objectif en en trois sous-espaces disjoints :

- *L'espace dominé* qui contient les solutions dominées par au moins une solution contenue dans l'archive (FIGURE 3a) ;
- *L'espace de diversification* qui contient les solutions qui ne sont dominées par aucune solution contenue dans l'archive et qui ne dominent aucune solution contenue dans l'archive (FIGURE 3b) ;
- *L'espace d'intensification* qui contient les solutions qui dominent au moins une solution contenue dans l'archive (FIGURE 3c).

La taille de l'espace dominé par une archive est un indicateur, nommé *hypervolume* $\mathcal{H}$ [23], communément utilisé pour mesurer la qualité d'une archive. Plus l'hypervolume d'une archive est grand, plus sa qualité est élevée. Clairement, la taille de l'espace dominé ne peut être augmentée qu'en ajoutant dans l'archive des solutions provenant de l'espace d'intensification ou de

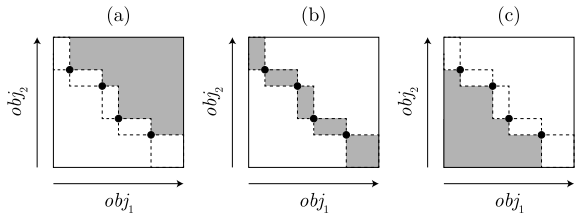


FIGURE 3 – Une archive partitionne l'espace objectif en trois sous-espaces disjoints : (a) l'espace dominé, (b) l'espace de diversification et (c) l'espace d'intensification.

l'espace de diversification. Observons que, l'archive devant rester sans dominance, toutes les solutions dominées par une nouvelle solution (provenant de l'espace d'intensification) doivent être supprimées de l'archive.

3 Travaux similaires

La contrainte epsilon (ou ϵ-constraint) est certainement la méthode la plus communément utilisée pour résoudre les problèmes d'optimisation multiobjectif au moyen de la programmation par contraintes [6, 21]. L'idée est de décomposer le problème original en une séquence de sous-problèmes à un seul objectif. A chaque itération, un nouveau sous-problème est généré en contraignant tous les objectifs, sauf un, à prendre une meilleure valeur que celle assignée dans la solution optimale du sous-problème précédent. A terme, l'ensemble de ces solutions optimales forme le front de Pareto du problème original.

La contrainte de Pareto [5, 7, 17] est une alternative (plus efficace [5]) pour calculer le front de Pareto de manière exacte. L'idée derrière cette contrainte est d'utiliser une archive pour élaguer les branches de l'arbre de recherche qui ne peuvent mener à une solution qui ne soit pas dominée par au moins une solution contenue dans l'archive (i.e. la solution ne peut provenir de l'espace dominé par l'archive). Chaque fois qu'une nouvelle solution est trouvée, celle-ci est ajoutée dans l'archive pour en améliorer sa qualité. La qualité de l'archive continue de s'améliorer incrémentalement tout au long de la recherche jusqu'à devenir le front de Pareto dont l'optimalité est prouvée lorsque la recherche est complète. Formellement, la contrainte de Pareto impose la proposition suivante où $\mathcal{A}$ est l'archive améliorée incrémentallement :

$$\bigwedge_{sol \in \mathcal{A}} \bigvee_{i=1}^{m} obj_i < sol_i \tag{4}$$

4 Recherche Multiobjectif à voisinage large

La recherche à voisinage large hybride la programmation par contraintes et la recherche locale pour améliorer incrémentallement une meilleure solution courante. A chaque itération, un voisinage de cette solution est généré en relachant une partie de sa structure (e.g. la valeur assignée à une partie des variables de décisions). La programmation par contraintes est utilisée pour explorer ce voisinage en résolvant le sous-problème généré par l'étape de relaxation.

Au lieu de se concentrer sur une seule solution, la recherche multiobjectif à voisinage large (MO-LNS) se concentre sur une population de meilleure solutions courantes i.e. une archive. A chaque itération, une solution de l'archive est sélectionnée, relâchée, et réoptimisée pour améliorer strictement la qualité de cette archive. Comme précisé ci-dessus, il existe deux manières d'améliorer la qualité d'une archive :

— *Diversifier l'archive* en y ajoutant une solution provenant de son espace de diversification. La solution peut être ajoutée directement dans l'archive puisque celle-ci ne domine aucune solution contenue dans l'archive ;

— *Intensifier l'archive* en y ajoutant une solution provenant de son espace d'intensification. Dans ce cas, il est nécessaire de supprimer toutes les solutions de l'archive qui sont dominées par la nouvelle solution avant de l'ajouter (l'archive doit rester sans dominance).

En utilisant la contrainte de Pareto (posée sur l'archive à améliorer), nous nous assurons qu'aucune solution appartenant à l'espace dominée ne puisse être découverte.

4.1 Sélectionner la solution à relacher

Le choix de la solution à relâcher lors de chaque itération peut avoir un impact important sur les chances de succès de l'itération i.e. les chances d'améliorer l'archive. La recherche de nouvelles solutions ayant lieu dans le voisinage de la solution sélectionnée, il y a de fortes chances pour que ces nouvelles solutions soient proches (dans l'espace objectif) de la solution à l'origine du voisinage. Nous appelons ce phénomène *effet de proximité*.

Il est reconnu par la communauté multiobjectif qu'une archive de faible cardinalité bien répartie sur le front de Pareto est préférable à une archive de forte cardinalité éloignée du front ou concentrée dans une certaine partie. Dès lors, il semble intuitif qu'une heuristique de sélection aléatoire pousse l'archive à se diversifie. Malheureusement, cette stratégie peut avoir un impact fortement négatif du à l'effet de proximité.

Imaginons qu'à un certain point, l'archive contienne un groupement de solutions. Sélectionner une solution de ce groupement tendra à générer de nouvelles solutions proche de celle-ci et donc à renforcer le groupement. Dès lors, la probabilité de sélectionner une solution de ce même groupement est renforcée et ainsi de suite. Pour éviter ce problème, nous suggérons une simple heuristique aléatoire qui tend à remplir les "trous" dans l'archive. L'idée est de sélectionner un point aléatoirement sur l'hyperplan passant par les extrémitées de l'archive (i.e. une ligne dans le cas d'un problème biobjectif). La solution sélectionnée est celle qui se trouve le plus proche dans l'espace objectif en terme de distance Euclidienne (voir FIGURE 4). Dans la suite, nous faisons référence à cette stratégie par *heuristique du plus proche voisin*.

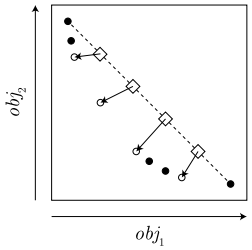


FIGURE 4 – L'heuristique du plus proche voisin tend à sélectionner les solutions proches de "trous" dans l'archive. La ligne discontinue correspond à l'hyperplan entre les extrémités de l'archive. Les losanges correspondent à de possibles points générés aléatoirement sur cet hyperplan. Chaque flêche pointe vers la solution la plus proche de ces points.

4.2 Guider l'intensification et la diversification

Une archive contenant de nombreuses solutions éloignées du front de Pareto est la conséquence d'une diversification trop importante au début de la recherche. Cela peut-être du à de nombreux facteur tels que la nature des voisinages explorés ou tout simplement l'heuristique utilisée pour explorer l'arbre de recherche. Afin de contrer ce problème, il est souvent intéressant de trouver rapidement un petit nombre de solutions proches du front de Pareto avant de diversifier l'archive. A l'instar du framework de recherche à voisinage large avec objectifs variables [16], nous proposons de controler dynamiquement les bornes des différents objectifs pour forcer l'intensification de l'archive durant la recherche. Trois modes de contrôle sont utilisés :

— *Sans filtrage* : les bornes de la variable objectif ne sont pas contraintes à être améliorées ;
— *Filtrage faible* : chaque fois qu'une nouvelle solution est découverte, la borne supérieure de la variable objective est contrainte à être plus petite ou égale à la valeur assignée dans cette solution i.e. $obj_i \leq sol_i$;
— *Filterage fort* : chaque fois qu'une nouvelle solution est découverte, la borne supérieure de la variable objectif est contrainte à être strictement plus petite que la valeur assignée dans cette solution i.e. $obj_i < sol_i$.

4.2.1 Diversification

Le but d'une phase de diversification est de compléter l'archive de nouvelle solutions non-dominées sans pour autant dominer les solutions contenues dans l'archive. Pour ce faire, nous configurant toutes les variables objectif en mode sans filtrage. Ainsi, nous laissons la contrainte de Pareto s'assurer que les nouvelles solutions découvertes n'appartiennent pas à l'espace dominée par l'archive.

4.2.2 Intensification

Le but d'une phase d'intensification est de découvrir de nouvelles solutions que domine la solution relâchée. Nous proposons deux différentes configuration des objectifs permettant de forcer ce résultat :

— *Intensification stricte* : tous les objectifs sont configuré en mode de filtrage fort ;
— *Intensification guidée* : tous les objectifs sont configuré en mode de filtrage faible à l'exception d'un objectif configuré en mode de filtrage fort (cet objectif guide l'intensification).

5 Résultats

Cette section compare les performances de la recherche multiobjectif à voisinage large avec celle d'une approche en programmation par contraintes utilisant uniquement la contrainte de Pareto. Nous présentons les résultats obtenus sur le problème biobjectif du sac à dos ainsi que sur une version biobjectif du problème d'allocation de conteneurs. Toutes nos expériences ont été réalisées au moyen du solver open-source OscaR[11] sur une machine équipée d'un processeur Intel Core i7 cadencé à 2.7GHz.

5.1 Problème du sac à dos à deux objectifs

Nos expériences furent réalisées sur plusieurs instances du problème du sac à dos allant de 50 à 250 objets (ces instances proviennent de la bibliothèque MOCOLib [22]). Les résultats présentés dans la TABLE 5.1 correspondent à la moyenne des résultats obtenus au moyen de 10 seeds différentes et d'une limite de temps fixée à une minute. La taille et l'hypervolume du front

			$\mathcal{H}_S$		$\lvert S\rvert$		$\lvert S \cap S^*\rvert$	
Inst.	$\lvert S^*\rvert$	$\mathcal{H}_{S^*}$	MO-LNS	CP	MO-LNS	CP	MO-LNS	CP
100A	172	15,59	**15,59**	15,05	**172**	128	**172**	112
100B	174	15,12	**15,12**	14,62	**170,6**	124	**164,7**	93
100C	64	16,68	**16,68**	**16,68**	**64**	**64**	**64**	**64**
100D	76	16.31	**16.31**	16.28	**76**	73	**76**	73
150A	244	39.66	**39.66**	35.42	**226.2**	88	**187**	31
150B	348	41.46	**41.46**	36.31	**303.3**	91	**192.4**	51
150C	166	34.17	**34.17**	33.15	**155.6**	83	**127**	34
150D	207	36.04	**36.04**	32.88	**199.8**	88	**155.6**	62
200A	439	64.34	**64.34**	57.43	**361**	86	**178.2**	34
200B	397	65.78	**65.77**	58.82	**345.2**	108	**232.8**	54
200C	328	57.48	**57.46**	48.30	**297.8**	60	**187.1**	18
200D	361	73.42	**73.40**	62.71	**304.1**	64	**176.9**	29
250A	629	94.37	**94.34**	78.68	**433.5**	70	**95.5**	12
250B	629	89.67	**89.65**	74.81	**410.9**	90	**107.9**	44
250C	528	91.25	**91.24**	75.30	**383.3**	72	**108.9**	22
250D	424	66.56	**66.55**	56.98	**303.4**	51	**142.8**	26

TABLE 1 – Comparison of MO-CP and MO-LNS on standard instances of the Bi-Objective Knapsack Problem.

de Pareto exacte est donnée dans les colonnes 2 et 3. Les hypervolumes obtenus par MO-LNS et CP sont donnés dans les colonnes 4 et 5 alors que la taille des archives correspondantes sont données dans les colonnes 6 et 7. Finalement, les colonnes 8 et 9 présentent le nombres de solutions efficaces trouvées par chaque méthode. Comme nous pouvons le voir, MO-LNS obtient de bien meilleur résultats. Aussi, les valeurs d'hypervolume obtenues par MO-LNS sont très proches de celles du front optimal.

5.2 problème biobjectif d'allocation de conteneurs

Le problème d'allocation de conteneurs consiste à assigner différents volumes de produits chimiques dans différents conteneurs tout en respectant un ensemble de contraintes de sécurité e.g. éviter de placer des produits dont la réaction est dangereuse dans des conteneurs proches. Une assignation idéale doit maximier le volume de conteneurs non utilisé afin de minimiser les couts de nettoyage. Minimiser le nombre de conteneurs utilisé est également souhaitable pour maximiser les chances d'accepter de nouvelles cargaison par la suite.

Les résultats obtenus sur ce problème sont présenté dans la TABLE 5.2 dont les colonnes ont la même signification que dans la table précédente. Le front de Pareto exact de chaque instance a été calculer au moyen de la contrainte epsilon et du solver MIP Gurobi avec un temps de calcul d'environ 3 minutes. Le framework MO-LNS est en mesure de trouver ces front optimaux en moins d'une minute. Notons également que l'approche en pure programmation par contrainte n'est capable de ne trouver qu'une seule solutio efficace en

			$\mathcal{H}_S$		$\lvert S\rvert$		$\lvert S \cap S^*\rvert$	
Inst.	$\lvert S^*\rvert$	$\mathcal{H}_{S^*}$	MO-LNS	CP	MO-LNS	CP	MO-LNS	CP
A	6	1848	**1848**	1022	**6**	4	**6**	1
B	7	2976	**2976**	1010	**7**	3	**7**	0
C	8	3597	**3597**	852	**8**	2	**8**	0
D	8	5358	**5358**	555	**8**	1	**8**	0

TABLE 2 – Comparison of MO-CP and MO-LNS on real-life instances of the Bi-Objective Tank Allocation Problem.

5 minutes (instance A). Aussi, l'hypervolume montre que les solutions trouvée via pure programmation par contraintes sont fort éloignées du front de Pareto.

6 Conclusion

Cet article décrit une manière simple d'étendre le framework de recherche à voisinage large pour résoudre les problèmes d'optimisation multiobjectif. Ce nouveau framework utilise la contrainte de Pareto pour améliorer incrémentalement la qualité de l'approximation du front de Pareto. Afin d'accélérer la convergence vers les solutions de qualité, nous proposons de mixer des phases d'intensification et de diversification. Nos résultats démontre que cette approche incomplète propose des résultats bien supérieurs à ceux obtenus via des approches classique en pure programmation par contraintes.

Références

[1] 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, May 2004.

[2] Juan J. Durillo and Antonio J. Nebro. jmetal : A java framework for multi-objective optimization. *Advances in Engineering Software*, 42 :760–771, 2011.

[3] Matthias Ehrgott. *Multicriteria optimization*, volume 2. Springer Berlin, 2005.

[4] Matthias Ehrgott and Xavier Gandibleux. Hybrid metaheuristics for multi-objective combinatorial optimization. *Hybrid metaheuristics*, pages 221–259, 2008.

[5] M. Gavanelli. An algorithm for multi-criteria optimization in csps. *ECAI*, 2 :136–140, 2002.

[6] Yacov Y Haimes, Leon S Lasdon, and David A Wismer. On a bicriterion formulation of the problems of integrated system identification and system optimization. *IEEE Transactions on Systems, Man, and Cybernetics*, 1(3) :296–297, 1971.

[7] Renaud Hartert and Pierre Schaus. A Support-Based Algorithm for the Bi-Objective Pareto Constraint. In *28th AAAI Conference on Artificial Intelligence (AAAI 2014)*, 2014.

[8] Thibaut Lust and Jacques Teghem. Two-phase Pareto local search for the biobjective traveling salesman problem. *Journal of Heuristics*, 16(3) :475–510, 2010.

[9] Jean-Baptiste Mairy, Yves Deville, and Pascal Van Hentenryck. Reinforced adaptive large neighborhood search. In *8th Workshop on Local Search techniques in Constraint Satisfaction (LSCS 2011). A Satellite Workshop of CP*, Perugia, Italy, 2011.

[10] R Timothy Marler and Jasbir S Arora. Survey of multi-objective optimization methods for engineering. *Structural and multidisciplinary optimization*, 26(6) :369–395, 2004.

[11] OscaR Team. OscaR : Scala in OR, 2012. Available from `https://bitbucket.org/oscarlib/oscar`.

[12] Laurent Perron, Paul Shaw, and Vincent Furnon. Propagation guided large neighborhood search. *Principles and Practice of Constraint Programming–CP 2004*, pages 468–481, 2004.

[13] J.C. Régin. Global constraints and filtering algorithms. *Constraint and Integer Programming–Towards a Unified Methodology*, pages 89–129, 2003.

[14] 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, and cbls. *Constraints*, 16(2) :125–147, 2011.

[15] P. Schaus, P. Van Hentenryck, and A. Zanarini. Revisiting the soft global cardinality constraint. *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, pages 307–312, 2010.

[16] Pierre Schaus. Variable objective large neighborhood search : A practical approach to solve over-constrained problems. In *IEEE International Conference on Tools with Artificial Intelligence (ICTAI)-2013*, 2013.

[17] Pierre Schaus and Renaud Hartert. Multi-Objective Large Neighborhood Search. In *19th International Conference on Principles and Practice of Constraint Programming*, 2013.

[18] Pierre Schaus, Jean-Charles Régin, Rowan Van Schaeren, Wout Dullaert, and Birger Raa. Cardinality reasoning for bin-packing constraint : application to a tank allocation problem. In *Principles and Practice of Constraint Programming*, pages 815–822. Springer, 2012.

[19] P. Shaw. Using constraint programming and local search methods to solve vehicle routing problems. *Principles and Practice of Constraint Programming–CP98*, pages 417–431, 1998.

[20] W.J. van Hoeve, G. Pesant, and L.M. Rousseau. On global warming : Flow-based soft global constraints. *Journal of Heuristics*, 12(4) :347–373, 2006.

[21] Luc N. Van Wassenhove and Ludo F. Gelders. Solving a bicriterion scheduling problem. *European Journal of Operations Research*, 4 :42–48, 1980.

[22] Xavier Gandibleux. A collection of test instances for multiobjective combinatorial optimization problems, 2013. Available from `http://xgandibleux.free.fr/MOCOlib/`.

[23] Eckart Zitzler, Lothar Thiele, Marco Laumanns, Carlos M Fonseca, and V Grunert da Fonseca. Performance assessment of multiobjective optimizers : An analysis and review. *Evolutionary Computation, IEEE Transactions on*, 7(2) :117–132, 2003.

Adaptation de la méthode Embarrassingly Parallel Search pour un centre de calcul

Mohamed Rezgui * **Jean-Charles Régin** * **Arnaud Malapert** *

Univ. Nice Sophia Antipolis, CNRS, I3S, UMR 7271, 06900 Sophia Antipolis, France
rezgui@i3s.unice.fr, {Jean-Charles.REGIN,Arnaud.Malapert}@unice.fr

Résumé

Nous proposons une adaptation de la méthode Embarrassingly Parallel Search (EPS) pour les centres de calcul. EPS est une méthode simple et efficace pour la résolution parallèle des problèmes en PPC. EPS décompose statiquement le problème en un grand nombre de sous-problèmes distincts qui sont résolus par des *workers* (threads/processus). Le nombre de sous-problèmes est proportionnel au nombre de cœurs. EPS obtient d'excellents résultats sur des machines multi-cœurs (40), mais ceux-ci se dégradent sur un centre de calcul (512 cœurs). Dans cet article, nous montrons que cette dégradation est due à la décomposition et proposons une nouvelle décomposition parallèle pour remédier à ce problème. Finalement, nous montrons que les gains d'EPS sont supérieurs d'un ou plusieurs ordres de grandeur à ceux du *work stealing*.

1 Introduction

En programmation par contraintes, les solveurs utilisent le parallélisme pour obtenir des gains de performance en partageant des tâches durant la résolution. Plusieurs méthodes ont été proposées, la plus connue étant le *work stealing* [11, 13, 6, 15, 4, 9]. Récemment, une nouvelle approche, nommée Embarrassingly Parallel Search (EPS) [12], donnait de bons résultats.

L'idée d'EPS est de décomposer le problème initial en un grand nombre de sous-problèmes qui sont consistants avec la propagation (l'application du mécanisme de propagation sur ces problèmes ne détecte pas d'inconsistance). Ces sous-problèmes sont ajoutés dans une queue gérée par un *master*. Ensuite, chaque *worker* prend un sous-problème de la queue et le résout. On répète le processus jusqu'à que tous les sous-problèmes soient résolus. La sélection des sous-problèmes par les *workers* est dynamique et il n'y a aucune communication entre les *workers*. EPS est basé sur l'idée que s'il y a un grand nombre de sous-problèmes à résoudre alors les temps de résolution des *workers* seront bien équilibrés même si les temps de résolution des sous-problèmes sont déséquilibrés. Autrement dit, l'équilibrage de charge est automatiquement obtenu au sens statistique. Ici, le nombre de sous-problèmes ne dépend pas du problème initial mais plutôt du nombre de *workers*. Des expériences ont montré qu'une décomposition doit générer environ 30 sous-problèmes par *worker*.

Des expérimentations préliminaires de la méthode EPS sur un centre de calcul (512 cœurs/*workers*) ont montré qu'on obtient un bon passage à l'échelle de la résolution d'un problème sans tenir compte du temps de décomposition. Cependant, la décomposition devient plus difficile et la part de la décomposition dans le temps total augmente avec le nombre de sous-problèmes. Tout d'abord, le nombre de sous-problèmes à générer augmente de manière linéaire avec le nombre de cœurs. Ensuite, le temps total de résolution diminue lorsqu'on augmente le nombre de *workers*. Enfin, la décomposition d'EPS proposée dans [12] n'est pas parallélisée de manière efficace. Dans cet article, nous améliorons la décomposition parallèle du problème initial pou un grand nombre de *workers*.

Une décomposition parallèle naïve a été proposée dans [12]. Elle décompose le problème initial en un nombre de sous-problèmes égal au nombre de *workers* et on assigne un sous-problème à chaque *worker*. Ensuite, chaque *worker* décompose son sous-problème en 30 sous-problèmes. C'est suffisant lorsque le nombre de *workers* est limité (40 par exemple) mais cela devient problématique avec des centaines de *workers*. Ce gain est limité car il n'y a aucune raison d'avoir des sous-problèmes équivalents à décomposer. L'idéal serait d'avoir plus de sous-problèmes

*Ces travaux ont bénéficié d'un accès aux moyens de calculs et de visualisation du Centre de Calcul Interactif hébergé à l'Université Nice Sophia Antipolis. Ils sont supportés également par OSEO, avec le projet ISI "Pajero".

à décomposer mais avoir plus de sous-problèmes est ce qu'on cherche ! C'est la raison pour laquelle nous avons besoin de trouver un autre algorithme. Cependant, nous avons constaté que :

1. la différence entre les gains de performance diminue lorsque le nombre de sous-problèmes augmente. Cette évolution n'est pas linéaire. Il y a une grande différence entre les gains de performance lorsqu'il y a moins de 5 sous-problèmes par *worker*. Ces différences diminuent lorsqu'il y a plus de 5 sous-problèmes par *worker*.
2. une décomposition ne doit pas générer des sous-problèmes inconsistants, car une telle décomposition aurait un impact négatif sur les performances.
3. découper un problème initial en un petit nombre de sous-problèmes sans qu'on vérifie l'inconsistance prend un temps raisonnable par rapport au temps total de décomposition.

À partir de ces observations, nous devons trouver un processus itératif qui décompose le problème. Ainsi, nous proposons une méthode en 3 phases :

– une phase initiale où l'on génère 1 sous-problème par *worker* le plus rapidement possible sans utiliser la propagation qui détecte l'inconsistence des sous-problèmes.
– une phase principale génère 5 sous-problèmes par *worker*. Chaque sous-problème est consistant avec la propagation.
– une phase finale génère 30 sous-problèmes par *worker* à partir de l'ensemble des sous-problèmes calculés à la fin de la phase principale.

En générant un petit nombre de sous-problèmes, on risque d'avoir une répartition de charge non équilibrée. Cependant, le coût relatif est faible au début de la décomposition. Ainsi, il est raisonnable de décomposer rapidement pour "démarrer" les workers qui à leur tour vont redécomposer les sous-problèmes. Ensuite, il faut générer 5 sous-problèmes par *worker* à partir des sous-problèmes précédemment calculés afin que la répartition de charge devienne raisonnablement plus équilibrée. Par la suite, les workers redécomposent les sous-problèmes en 30 sous-problèmes par *worker*. Une fois la décomposition terminée, les workers résolvent des sous-problèmes.

Nous rappelons d'abord les préliminaires. Puis nous décrivons les décompositions existantes et nous présentons une parallélisation efficace de la décomposition. Finalement, nous donnons des résultats expérimentaux et concluons.

2 Préliminaires

Un *worker* est une unité de calcul chargée de résoudre un sous-problème. La plupart du temps, il correspond à un cœur d'un processeur. On considère qu'il y a w *workers*. Nous rappelons quelques définitions. Puis, nous présentons deux méthodes que nous comparerons.

2.1 Work stealing

Le *work stealing* a été proposé à l'origine par [3] et a été implémenté pour la première fois sur un machine parallèle Lisp [5]. Cette méthode découpe le problème dynamiquement durant la résolution.

Les *workers* résolvent des sous-problèmes et lorsqu'un *worker* termine un sous-problème, il "vole" un peu de travail d'un autre *worker*. Habituellement, il est mis en œuvre comme suit : quand un *worker* W n'a plus de travail, il demande du travail à un autre *worker* V. Si la réponse est positive, alors le *worker* V découpe le problème qu'il résout actuellement en deux sous-problèmes et en donne un au *worker* W. Par exemple, appliqué à un arbre binaire, si V est sur la branche de gauche, il donne la branche de droite à W sinon il descend d'un niveau et donne la branche de droite à W. Dans le cas où la réponse est négative, W demande à un autre *worker* U, jusqu'à ce qu'il arrive à avoir du travail ou qu'il ait demandé à tous les *workers*.

Cette méthode a été implémentée sur plusieurs solveurs (Comet [9] ou ILOG Solver [11] par exemple) et de différentes façons [13, 6, 15, 4] selon que le travail est centralisé ou non, de la décomposition de l'arbre de recherche ou de la méthode de communication entre les *workers*. Le *work stealing* essaie de résoudre en partie le problème de l'équilibrage de la charge de travail en décomposant dynamiquement les sous-problèmes.
Quand un *worker* est en attente, il ne doit pas "voler" trop de problèmes faciles, car il devra redemander du travail presque immédiatement. Cela se produit fréquemment à la fin de la recherche lorsqu'un grand nombre de *workers* n'ont plus de sous-problèmes à résoudre et demandent tout le temps du travail et ce qui entraine beaucoup de communications inutiles.

Ainsi, la méthode obtient généralement un bon passage à l'échelle pour un petit nombre de *workers* mais il est difficile de maintenir un gain linéaire lorsque le nombre de *workers* augmente. Dans [2], la résolution en parallèle du problème des 17-reines ne passe pas à l'échelle à partir de 29 *workers*.

Certaines méthodes [14, 6] ont été développées pour tenter d'améliorer ce passage à l'échelle [14]. Xie et autres [14] proposent d'utiliser une approche impliquant plusieurs *masters* et *workers*. Chaque *master* possède ses *workers*. L'espace de recherche est divisé entre les différents *masters*, tel que chaque *master* met ses sous-arbres de recherche dans une queue qui seront distribués aux *workers*. Lorsqu'un nœud du sous-arbre est détecté comme un nœud racine d'un large sous-arbre, les *workers* génèrent un grand nombre de sous-arbres et l'ajoute dans la queue afin d'avoir

une meilleure répartition de charge.

Dans [6], les auteurs expérimentent sur 64 cœurs en utilisant le *work stealing*. Un *master* centralise toutes les informations (bornes, solutions et requêtes). Le *master* estime quel *worker* possède la plus grande quantité de travail afin de redistribuer le travail aux *workers* en attente.

Un autre inconvénient du *work stealing* qui n'est souvent pas mentionné est que sa mise en œuvre est intrusive. Elle est ainsi fortement dépendante du solveur et oblige d'avoir une très bonne connaissance du solveur CP et d'avoir accès à des fonctions internes. Il existe des méthodes qui tentent de répondre à ce problème [8].

2.2 EPS

L'Embarassingly Parallel Search (EPS) est décrit dans [12]. Cette méthode décompose de manière statique le problème initial en un grand nombre de sous-problèmes consistants avec la propagation qui sont ensuite mis dans une queue à la disposition des *workers*. Lorsque la décomposition est terminée, les *workers* prennent dynamiquement les sous-problèmes de la queue lorsqu'ils sont en attente. Précisément, EPS est basée sur les étapes suivantes :

- on découpe le problème en p sous-problèmes tel que $p \geq w$ et on les met dans la queue.
- chaque *worker* prend à tour de rôle un sous-problème dans la queue et le résout.
- un *master* contrôle les accès concurrents de la queue.
- la résolution est terminée lorsque tous les sous-problèmes sont résolus.

Dans le cas des problèmes d'optimisation, le *master* gère la valeur de l'objectif. Lorsqu'un *worker* prend un sous-problème de la queue, il prend également la meilleure valeur courante de l'objectif. Ensuite, lorsqu'un *worker* résout le sous-problème, il communique au *master* la valeur de l'objectif trouvée. Il faut noter qu'il n'y a pas d'autres communications lorsqu'un *worker* trouve la meilleure solution, les autres *workers* qui sont en train de travailler ne peuvent pas l'utiliser pour améliorer leur solution courante.

La réduction des communications est un avantage par rapport au *work stealing*. Un autre avantage est la possibilité de rejouer la résolution en mémorisant les sous-problèmes et leur ordre de traitement. Cela ne coûte presque rien et c'est très utile dans le débogage d'applications.

2.3 Définitions

Un réseau de contraintes $\mathcal{CN} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ est défini de la manière suivante :

- un ensemble de n *variables* $\mathcal{X} = \{x_1, \ldots, x_n\}$
- un ensemble de n *domaines* finis $\mathcal{D} = \{D(x_1), \ldots, D(x_n)\}$ où $D(x_i)$ l'ensemble des *valeurs* possibles pour la variable x_i,
- un ensemble de *contraintes* entre les variables $\mathcal{C} = \{C_1, \ldots, C_e\}$. Une contrainte C_i est définie sur un sous-ensemble de variables $X_{C_i} = \{x_{i_1}, \ldots, x_{i_j}\}$ de $\mathcal{X}$ par un sous-ensemble du produit cartésien $D(x_{i1}) \times \ldots \times D(x_{ij})$, qui spécifie quelles combinaisons de valeurs des variables $\{x_{i_1}, \ldots, x_{i_j}\}$ admissibles entre elles.

Chaque contrainte C_i est associée à un algorithme de filtrage qui supprime les valeurs des domaines de ses variables pour lesquelles il n'est pas possible de satisfaire la contrainte. La propagation de contraintes applique les algorithmes de filtrage des contraintes de $\mathcal{C}$ dont les variables ont vu leur domaine réduit, à tour de rôle, jusqu'à ce que plus aucune réduction ne soit faite. On délègue la propagation de contraintes à un solveur externe. Par souci de commodités, nous utiliserons le mot "problème" à la place d'un réseau de contraintes lorsqu'on l'utilise pour représenter le réseau de contraintes et non pour la recherche d'une solution. Nous admettons que le problème P est consistant à la propagation si et seulement si l'exécution du mécanisme de propagation sur P ne déclenche pas un échec.

Notation 1 *Soit Q un problème, nous notons $D(Q, x)$ le domaine résultant de la variable x lorsqu'on applique le mécanisme de propagation sur Q.*

3 Décomposition en sous-problèmes

La méthode EPS est basée sur la décomposition du problème initial en q sous-problèmes. [12] a montré que les sous-problèmes générés doivent être consistants avec la propagation. Dans le cas contraire, l'algorithme parallèle peut considérer les sous-problèmes qui ne seraient pas résolus par un algorithme de recherche séquentiel.

Si on souhaite générer p sous-problèmes alors on peut utiliser l'algorithme suivant, nommé SIMPLEDECOMPOSITIONMETHOD. On considère un ordre quelconque des variables [1] $x_1, ..., x_n$. On calcule la valeur k tel que $|D(x_1)| \times \ldots \times |D(x_{k-1})| < p \leq |D(x_1)| \times \ldots \times |D(x_{k-1})| \times |D(x_k)|$. Puis, on génère toutes les affectations des variables de x_1 à x_k et on les regroupe s'il y en a beaucoup.

Une première parallélisation de cette décomposition est décrite dans [12]. Le problème initial est découpé en w sous-problèmes par la méthode de décomposition simple. Chaque *worker* reçoit un de ces sous-problèmes et le décompose en p/w sous-problèmes consistants avec la propagation en appliquant la décomposition séquentielle. Le *master* rassemble tous les sous-problèmes calculés. Si un *worker* n'arrive pas à générer p/w sous-problèmes, le *master* demande aux autres *workers* de redécomposer leurs sous-problèmes jusqu'à atteindre le nombre de sous-problèmes souhaité.

1. Dans cet article, on n'étudie pas l'influence d'un ordre spécifique.

Algorithm 1: Quelques fonctions utiles

SIMPLEDECOMPOSITION($\mathcal{CN}, p$)

calculer la valeur k tel que $|D(x_1)| \times \ldots \times |D(x_{k-1})| < p \leq |D(x_1)| \times \ldots \times |D(x_{k-1})| \times |D(x_k)|$;
génère toutes les affectations des variables de x_1 à x_k ;
regrouper ces affectations et mettre les sous-problèmes résultants dans S ;
retourner S et la valeur k;

COMPUTEDEPTH($\mathcal{CN}, cardS, \delta, p$)

retourner d tel que $cardS \times |D(x_{\delta+1})| \times \ldots \times |D(x_{d-1})| < p \leq cardS \times |D(x_{\delta+1})| \times \ldots \times |D(x_{d-1})| \times |D(x_d)|$;

GETDOMAINS(S)

retourner l'ensemble des domaines de $\mathcal{D} = \{D(S, x_1), D(S, x_2), \ldots, D(S, x_n)\}$ tel que $\forall x \in \mathcal{X}\ D(S, x) = \cup_{P \in S} D(P, x)$;

GENERATESUBPROBLEMS($\mathcal{CN}, S, d$)

lancer une recherche de solutions basée sur une DBDFS avec d la profondeur limite sur le réseau de contrainte formé par $\mathcal{CN}$ et la table de contraintes définie à partir des éléments de S;
retourner l'ensemble des feuilles qui ne sont pas des échecs;

4 Une décomposition parallèle efficace

Comme mentionné dans l'introduction, il y a trois observations majeures par rapport à la décomposition :

1. La différence de travail entre les *workers* décroit lorsque le nombre de sous-problèmes augmente. Ce n'est pas une relation linéaire. Il y a une grande différence entre les temps d'activité des *workers* lorsqu'il y a moins de 5 sous-problèmes par *worker*. Cette différence décroit lorsqu'il y a plus de 5 sous-problèmes par *worker*.
2. Une décomposition simple en sous-problèmes qui peuvent être inconsistants, génère rapidement des sous-problèmes car les inconsistances sont rapidement détectées.
3. La décomposition du problème initial en un petit nombre de sous-problèmes exige peu de temps par rapport à la durée totale de la résolution.

La méthode est décomposée en 3 principales phases. Une première phase rapide où le but est de donner du travail aux *workers* en ne considérant pas l'équilibrage de charge. Ensuite, une seconde phase qui est le cœur de la décomposition. Dans cette phase, on essaie de progresser dans la décomposition et de gérer la répartition de charge entre les *workers*, car on ne peut pas éviter son déséquilibre. On progresse par des courtes étapes de décomposition. Ces étapes sont suivies par la synchronisation des *workers* et par l'union des ensembles des sous-problèmes calculés par chaque *worker*. On veut ainsi régler le déséquilibre de la répartition de charge. En d'autres termes, les *workers* décomposent n'importe quel sous-problème en un petit nombre de sous-problèmes. Ensuite, on rassemble tous les sous-problèmes et chaque sous-problème est à nouveau décomposé en un petit nombre de sous-problèmes. Lorsque le nombre de sous-problèmes générés est proche de 5 sous-problèmes par *worker*, on sait qu'il faut avoir moins de sous-problèmes pour l'équilibrage de charge. Ainsi, on peut continuer et aller à la dernière phase : la décomposition des sous-problèmes permettant d'atteindre 30 sous-problèmes par *worker*.

Les phases sont définies de la manière suivante :

- Une phase initiale dans laquelle on décompose le plus rapidement possible le problème en autant de sous-problèmes qu'il y a de *workers*
- Une phase principale qui a pour but de générer 5 sous-problèmes par *worker*. Chaque sous-problème est consistant avec la propagation. Cette phase est divisée en plusieurs étapes afin d'atteindre ce but tout en équilibrant le travail entre les *workers*.
- Une phase finale qui consiste à générer 30 sous-problèmes par *worker* à partir de l'ensemble des sous-problèmes calculés par la phase principale.

L'algorithme 2 est une possible implémentation de la nouvelle décomposition parallèle. Cet algorithme utilise nbPbStep, un tableau de valeurs définissant le nombre de sous-problèmes requis pour chaque étape.

La question restante est la définition du nombre d'étapes et le nombre de sous-problèmes par *worker* qui doivent être générés pour chaque étape de la seconde phase. La décomposition de la première phase est clairement mauvaise et on doit s'arrêter pour redistribuer les sous-problèmes aux *workers* le plus rapidement possible. Les expérimentations ont montré qu'il faut s'arrêter lorsqu'on génère 1 sous-problème consistant avec la propagation par *worker*. Ensuite, un arrêt à 5 sous-problèmes par *worker* est suffisant pour la seconde phase.

5 Partie Expérimentale

Machines Toutes les expériences ont été effectuées sur le Centre de Calculs Interactifs de l'Université de Nice Sophia Antipolis ayant 1152 cœurs répartis sur 144 processeurs Intel E5-2670 (chaque processeur possède 16 cœurs), avec 4608 Go de mémoire au total et fonctionnant sous Linux (`http://calculs.unice.fr/`). Le centre de calcul utilise un outil de planification (OAR) qui s'occupe de la gestion et de la soumission des travaux (l'exécution des programmes).

Algorithm 2: EPS : Amélioration de la décomposition en parallèle

```
WorkerDec(CN, Q, d)
    S ← ∅;
    run in parallel
        while Q ≠ ∅ do
            prendre P ∈ Q et supprimer P dans Q;
            S' ←GenerateSubproblems(CN, P, d);
            S ← S ∪ S'
    return S;
Decompose(CN, S, nbspb)
    while |S| < nbspb do
        d ←ComputeDepth(CN, |S|, d, nbspb);
        S ←WorkerDec(CN, S, d);
        if S = ∅ then return ∅;
        CN ← (X, GetDomain(S), C)
    return S;
ParallelDecomposition(CN, nbPbStep, numStep, p)
    (S, d) ←SimpleDecomposition(CN, nbPbStep[0]);
    S ←WorkerDec(CN, S, d);
    CN ← (X, GetDomain(S), C);
    for i=0 to numStep-1 do
        S ←Decompose(CN, S, nbPbStep[i]);
        if S = ∅ or |S| ≥ p then return S;
    return S;
```

Détails d'implémentation EPS est implémentée dans gecode 4.0.0 [1]. Nous utilisons MPI (Message Passing Interface), un protocole de communication pour échanger les informations entre les processus. Chaque processus lit un modèle FlatZinc pour initialiser le problème et les différentes informations (sous-problèmes, résultats) sont échangées à travers les messages entre le *master* et les *workers*.

Instances sélectionnées 38 instances sont sélectionnées. Deux types de problème sont étudiés : les problèmes d'énumération et les problèmes d'optimisation. Certaines viennent de la CSPLib et ont été modélisées par Håkan Kjellerstrand (voir [7]) et d'autres proviennent de la distribution minizinc (1.5 voir [10]).
Pour étudier la décomposition, on sélectionne les instances utilisées dans [12] qui ne sont pas trop longues à résoudre avec le solveur Gecode ainsi que les autres instances considérées difficiles (plus de 500 secondes et moins de 1h) afin de les utiliser pour les expérimentations avec un plus grand nombre de *workers* (512 *workers*).

Tests Premièrement, il est important de noter que la décomposition doit être terminée pour démarrer la résolution des *workers*. On note respectivement t_{dec} et t_{res} le temps de décomposition et le temps de résolution des *workers*. Ainsi, le temps total de la résolution t est égal à $t_{dec} + t_{res}$. Soit t_0 le temps de la résolution séquentielle, su donne le gain de performance par rapport au temps de la résolution séquentielle tel que $su = t_0 \div t$. De plus, on définit su_{res} le gain de performance de la résolution des *workers* tel que $su = t_0 \div t_{res}$.

5.1 Analyse sur la décomposition

5.1.1 Profondeur dans la décomposition

La figure 1 montre l'évolution de la profondeur lorsqu'on utilise plus de *workers* avec EPS. Comme prévu, la profondeur dans la décomposition croît avec le nombre de *workers*. Parfois, il peut être important avec certaines instances, comme *warehouses* ou *talent scheduling film116* ou non comme *allinterval 15* (voir tableau 2). Ainsi, l'amélioration de la décomposition peut avoir un impact sur t, notamment pour des problèmes faciles.

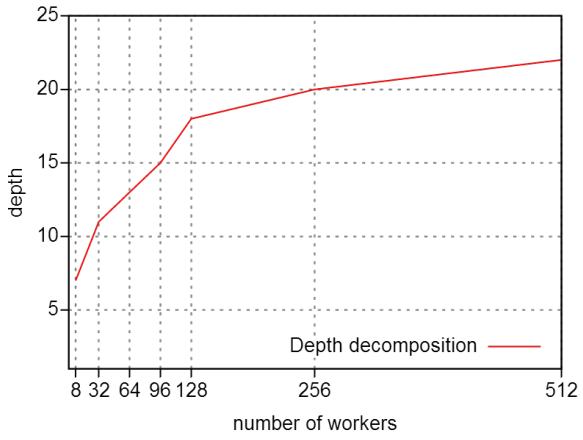


FIGURE 1 – Profondeur pour atteindre 30 sous-problèmes par *worker* en fonction du nombre de *workers*.

5.1.2 Les paramètres de la décomposition

Maintenant, on doit choisir nbPbStep[1] pour la phase principale. On sélectionne quelques instances représentatives afin de fixer le bon nombre de sous-problèmes par *worker*. Le tableau 1 montre que 5 est un bon choix.

5.1.3 Comparaison des algorithmes de décomposition

On note respectivement Dec_{seq}, $Dec_{//1}$ et $Dec_{//2}$ la décomposition séquentielle et la décomposition parallèle

Instance	nbPbStep[1]				
	3	4	5	6	7
sb_sb_13_13_6_4	0.8	0.5	0.7	1.1	1.0
sugiyama	2.5	1.4	1.3	1.5	1.5
patternSetMiningGerman	1.4	1.4	0.9	0.9	1.1
radiation_03	0.6	0.6	0.7	0.4	0.6
total time(s)	5.3	3.8	**3.6**	3.9	4.2

TABLE 1 – Comparaison des temps de décomposition (en secondes) en fonction d'un nombre de sous-problèmes fixé dans la seconde phase avec 96 *workers*.

naïve qui sont décrites dans [12] ainsi que la nouvelle décomposition parallèle. On peut remarquer que tous les algorithmes donnent toujours la même décomposition. Le tableau 2 compare le temps de décomposition entre les différentes méthodes de décomposition avec 96 *workers* et le facteur de gain su de chaque algorithme de décomposition. On note que $Dec_{//1}$ améliore Dec_{seq} par un facteur de gain de 3,8 et $Dec_{//2}$ améliore $Dec_{//1}$ par un facteur de gain de 3,4. Ainsi, la nouvelle méthode de décomposition parallèle améliore clairement la décomposition.

Instance	Seq.		$Dec_{//2}$		$Dec_{//1}$		Dec_{seq}	
	t_0	su_{res}	t_{dec}	su	t_{dec}	su	t_{dec}	su
	s	r	s	r	s	r	s	r
allinterval_15	220.0	83.5	0.3	75.1	0.6	69.1	3.1	38.2
magicsequence_40000	316.6	116.8	1.4	76.6	8.5	28.2	21.5	13.1
sportsleague_10	170.1	74.4	0.5	61.6	2.2	38.1	9.4	14.5
sb_sb_13_13_6_4	135.1	82.4	0.7	57.5	1.4	44.6	12.9	9.3
quasigroup7_10	287.0	76.5	0.6	65.6	3.8	38.1	13.1	17.0
non_non_fast_6	582.4	80.9	8.0	38.3	13.0	28.8	29.7	15.8
golombruler_13	1303.9	94.2	0.3	92.1	4.5	71.0	8.9	57.2
warehouses	139.4	219.9	0.7	108.3	4.9	25.1	14.8	9.1
setcovering	88.4	74.6	0.2	65.3	2.4	24.4	4.7	15.1
depot_placement_att48_5	113.3	74.9	0.4	60.3	1.4	38.9	6.2	14.7
depot_placement_rat99_5	20.3	76.9	0.4	30.9	0.4	32.1	6.9	2.9
fastfood_ff58	22.3	84.7	0.6	27.4	0.6	26.4	5.7	3.7
open_stacks_01_problem	99.1	78.1	0.3	62.1	1.9	31.4	14.1	6.4
open_stacks_01_wbp	180.7	72.2	0.9	53.0	3.2	31.9	21.7	7.5
sugiyama	237.5	80.5	1.3	56.5	5.0	29.7	33.0	6.6
patternSetMiningGerman	103.9	73.4	0.9	44.7	2.2	28.7	11.6	8.0
radiation_03	108.1	70.6	0.3	59.0	1.7	33.9	10.6	8.9
talent_scheduling_film	243.3	97.8	2.5	48.3	10.5	18.7	31.1	7.3
total(s) and geom. average(r)	4371.1	85.8	**20.2**	**57.0**	68.1	33.5	259.0	10.7

TABLE 2 – Comparaison des différents algorithmes de décomposition avec 96 *workers*.

5.1.4 Analyse du passage à l'échelle

On teste le passage à l'échelle d'EPS avec différents nombres de *workers* sur les instances difficiles. Les détails des facteurs de gain sont décrits dans le tableau 3. On remarque que la résolution du *Golomb-ruler* et des instances du *market-split* (environ 400 for 512 *workers*) passe très bien à l'échelle avec EPS.

5.2 Comparaison avec le *work stealing*

Le tableau 4 présente une comparaison entre EPS et le *work stealing* de gecode avec 512 *workers*. Le gain de performance moyen du *work stealing* est de 5,4. Plusieurs instances ne sont pas résolues avec 512 *workers*. Par contre, EPS atteint un gain de performance moyen de 223,9. La méthode EPS est meilleure que le *work stealing* sur toutes les instances sélectionnées. EPS arrive à obtenir de gains de performance presque linéaires avec 7 instances. Les autres instances étant booléennes, l'estimation de la profondeur d'EPS pour générer les sous-problèmes est mauvaise car elle s'appuie sur le produit des puissances de 2 (domaines booléens). Avec une mauvaise estimation, les synchronisations prennent du temps et par suite EPS ne parvient pas à

Instance	number of *workers*								
	$t_0(s)$	su							
	1w	8w	16w	32w	64w	96w	128w	256w	512w
market_split_s5-02	3314.4	7.3	14.2	25.4	50.7	69.7	101.5	201.7	405.9
market_split_u5-09	3266.6	7.3	14.3	25.7	51.5	68.6	103.0	207.4	411.8
market_split_s5-06	3183.9	6.4	12.7	24.0	48.0	64.0	96.0	197.5	384.0
prop_stress_0600	2729.2	3.8	6.7	16.1	24.1	32.2	48.3	104.2	193.1
nmseq_400	2505.8	4.1	7.2	15.0	30.1	40.1	60.1	117.7	240.4
prop_stress_0500	1350.6	2.5	4.4	13.1	20.2	26.9	40.4	81.8	161.6
fillomino_18	763.9	2.4	5.1	11.4	18.8	25.1	37.7	72.4	150.7
steiner-triples_09	604.9	5.7	12.3	21.7	41.5	55.3	83.0	143.2	332.0
nmseq_300	555.3	2.4	5.1	8.2	16.5	21.9	32.9	69.3	131.7
golombruler_13	1303.9	7.3	14.7	27.3	53.7	89.5	117.4	213.1	427.9
cc_base_mzn_rnd_test.11	3279.5	1.7	5.1	8.9	14.3	20.4	30.6	59.7	122.6
ghoulomb_3-7-20	2993.8	2.3	3.9	8.2	17.4	21.8	32.8	76.3	131.1
pattern_set_mining_k1_yeast	2871.3	2.9	5.8	11.5	23.9	30.5	45.8	91.6	183.2
still_life_free_8x8	2808.9	2.6	6.4	10.4	20.9	27.8	41.7	83.5	166.9
bacp-6	2763.3	6.7	12.1	23.7	47.4	63.1	94.7	212.4	378.9
depot_placement_st70_6	2665.1	3.4	7.3	14.7	29.4	39.2	58.8	147.6	235.1
open_stacks_01_wbp_20_20_1	1523.2	3.1	6.5	10.0	23.1	26.8	40.2	95.4	160.8
bacp-27	1499.7	5.6	11.2	20.4	43.8	54.4	81.6	214.3	326.5
still_life_still_life_9	1145.1	3.1	6.1	11.4	22.9	30.5	45.7	89.4	182.9
talent_scheduling_alt_film117	566.1	2.4	4.3	11.0	22.0	31.3	51.7	95.9	175.8
total (t_0) or geometric average (su)	41694.5	3.7	7.5	14.7	28.3	37.9	56.7	117.3	223.9

TABLE 3 – Facteurs de gain détaillés pour chaque instance et pour chaque nombre de *workers* avec EPS.

Instance	Seq.	Work stealing		EPS	
	t	t	s	t	s
market_split_s5-02	3314.4	-	-	8.2	**405.9**
market_split_u5-09	3266.6	-	-	7.9	**411.8**
market_split_s5-06	3183.9	-	-	8.3	**384.0**
prop_stress_0600	2729.2	1426.4	1.9	14.1	**193.1**
nmseq_400	2505.8	-	-	10.4	**240.4**
prop_stress_0500	1350.6	670.0	2.0	8.4	**161.6**
fillomino_18	763.9	-	-	5.1	**150.7**
steiner-triples_09	604.9	79.0	7.7	1.8	**332.0**
nmseq_300	555.3	-	-	4.2	**131.7**
golombruler_13	1303.9	15.5	83.9	3.0	**427.9**
cc_base_mzn_rnd_test.11	3279.5	-	-	26.8	**122.6**
ghoulomb_3-7-20	2993.8	575.4	5.2	22.8	**131.1**
pattern_set_mining_k1_yeast	2871.3	299.8	9.6	15.7	**183.2**
still_life_free_8x8	2808.9	1672.8	1.7	16.8	**166.9**
bacp-6	2763.3	330.1	8.4	7.3	**378.9**
depot_placement_st70_6	2665.1	1902.9	1.4	11.3	**235.1**
open_stacks_01_wbp_20_20_1	1523.2	153.9	9.9	9.5	**160.8**
bacp-27	1499.7	579.6	2.6	4.6	**326.5**
still_life_still_life_9	1145.1	140.1	8.2	6.3	**182.9**
talent_scheduling_alt_film117	566.1	95.5	5.9	3.2	**175.8**
total (t) ou moyenne geométrique (s)	41694.5	7941	5.4	195.7	**223.9**

TABLE 4 – Comparaison entre le *work stealing* et EPS avec 512 *workers*.

atteindre des bons gains de performance.

6 Conclusion

La décomposition séquentielle et la décomposition parallèle naïve d'EPS sont des méthodes acceptables quand il y a seulement un petit nombre de *workers*. Ces méthodes limitent les performances d'EPS sur un centre de calcul avec des centaines de cœurs. Dans cet article, nous avons décrit une version parallèle et efficace de la décomposition. En parallélisant la décomposition de problème et en fixant trois phases au cours du processus, EPS obtient une meilleure répartition de charge lors de la décomposition. Par conséquent, EPS passe à l'échelle avec un centre de calcul et obtient une accélération moyenne à $223,9$ avec gecode sur une machine avec 512 cœurs. On montre clairement que l'implémentation du *work stealing* dans le solveur gecode s'adapte mal sur des centaines de cœurs. EPS est plus efficace d'un ou plusieurs ordres de grandeur.

Références

[1] Gecode 4.0.0. http ://www.gecode.org/, 2012.

[2] Lucas Bordeaux, Youssef Hamadi, and Horst Samulowitz. Experiments with massively parallel constraint solving. In Craig Boutilier, editor, *IJCAI*, pages 443–448, 2009.

[3] F. Warren Burton and M. Ronan Sleep. Executing functional programs on a virtual tree of processors. In *Proceedings of the 1981 Conference on Functional Programming Languages and Computer Architecture*, FPCA '81, pages 187–194, New York, NY, USA, 1981. ACM.

[4] Geoffrey Chu, Christian Schulte, and Peter J. Stuckey. Confidence-based work stealing in parallel constraint programming. In Ian P. Gent, editor, *CP*, volume 5732 of *Lecture Notes in Computer Science*, pages 226–241. Springer, 2009.

[5] Robert H. Halstead, Jr. Implementation of multilisp : Lisp on a multiprocessor. In *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*, LFP '84, pages 9–17, New York, NY, USA, 1984. ACM.

[6] Joxan Jaffar, Andrew E. Santosa, Roland H. C. Yap, and Kenny Qili Zhu. Scalable distributed depth-first search with greedy work stealing. In *ICTAI*, pages 98–103. IEEE Computer Society, 2004.

[7] Håkan Kjellerstrand. http ://www.hakank.org/, 2014.

[8] Tarek Menouer, Bertrand Cun, and Pascal Vander-Swalmen. Partitioning methods to parallelize constraint programming solver using the parallel framework bobpp. 479 :117–127, 2013.

[9] Laurent Michel, Andrew See, and Pascal Van Hentenryck. Transparent parallelization of constraint programming. *INFORMS Journal on Computing*, 21(3) :363–382, 2009.

[10] MiniZinc. http ://www.g12.csse.unimelb.edu.au/minizinc/, 2012.

[11] Laurent Perron. Search procedures and parallelism in constraint programming. In Joxan Jaffar, editor, *CP*, volume 1713 of *Lecture Notes in Computer Science*, pages 346–360. Springer, 1999.

[12] Jean-Charles Régin, Mohamed Rezgui, and Arnaud Malapert. Embarrassingly parallel search. In Christian Schulte, editor, *Principles and Practice of Constraint Programming*, volume 8124, pages 596–610. Springer Berlin Heidelberg, 2013.

[13] Christian Schulte. Parallel search made simple. pages 41–57, 2000.

[14] Feng Xie and Andrew J. Davenport. Massively parallel constraint programming for supercomputers : Challenges and initial results. In Andrea Lodi, Michela Milano, and Paolo Toth, editors, *CPAIOR*, volume 6140 of *Lecture Notes in Computer Science*, pages 334–338. Springer, 2010.

[15] Peter Zoeteweij and Farhad Arbab. A component-based parallel constraint solver. In Rocco De Nicola, Gian Luigi Ferrari, and Greg Meredith, editors, *COORDINATION*, volume 2949 of *Lecture Notes in Computer Science*, pages 307–322. Springer, 2004.

Gestion du vidage de données satellite avec incertitude sur les volumes

Cédric Pralet[1] **Adrien Maillard**[1] **Gérard Verfaillie**[1]
Emmanuel Hébrard[2] **Nicolas Jozefowiez**[2] **Marie-Jo Huguet**[2]
Thierry Desmousceaux[3] **Pierre Blanc-Paques**[3] **Jean Jaubert**[4]

[1] Onera - The French Aerospace Lab, Toulouse
[2] LAAS/CNRS, Toulouse
[3] Airbus Defence and Space, Toulouse
[4] CNES, Toulouse
Prénom.Nom@onera.fr Prénom.Nom@laas.fr
Prénom.Nom@astrium.eads.net Prénom.Nom@cnes.fr

Résumé

Les satellites d'observation de la Terre sont des senseurs qui acquièrent des données, les compressent et les mémorisent à bord, puis les vident vers le sol. Du fait de l'utilisation d'algorithmes de compression de plus en plus sophistiqués, le volume de données résultant d'une acquisition est de moins en moins prédictible. Dans de telles conditions, planifier les activités de vidage du satellite hors-ligne au sol est de plus en plus problématique. Dans ce papier, nous rapportons les résultats d'une étude visant à évaluer le bénéfice d'une planification des activités de vidage à bord, quand le volume de données produit par chaque acquisition est connu.

Le problème de vidage des données à résoudre à bord est un problème d'affectation et d'ordonnancement avec des ressources non partageables, des contraintes de précédence, des durées minimum dépendantes du temps et un critère d'optimisation complexe. La librairie générique InCELL [10] est utilisée pour modéliser contraintes et critère, vérifier les contraintes non temporelles, propager les contraintes temporelles et évaluer le critère. Au dessus de cette librairie, des algorithmes de recherche gloutonne et locale ont été développés pour produire des plans de vidage, compte-tenu du temps et des ressources de calcul limités disponibles à bord.

Abstract

Earth observation satellites are space sensors which acquire data, compress and record it on board, and then download it to the ground. Because of the use of more and more sophisticated compression algorithms, the amount of data resulting from an acquisition is less and less predictable. In such conditions, planning satellite data download activities offline on the ground is more and more problematic. In this paper, we report the results of a work aiming at evaluating the positive impact of planning downloads onboard when the amount of data produced by each acquisition is known.

The data download problem to be solved on board is an assignment and scheduling problem with unsharable resources, precedence constraints, time-dependent minimum durations, and a complex optimization criterion. The generic InCELL library [10]) is used to model constraints and criterion, to check non temporal constraints, to propagate temporal constraints, and to evaluate the criterion. On top of this library, greedy and local search algorithms have been designed to produce download plans with limited time and computing resources available on board.

1 Introduction

Les satellites d'observation de la Terre sont des senseurs qui acquièrent des données, les compressent et les mémorisent à bord, puis les vident vers le sol. Du fait de l'utilisation d'algorithmes de compression de plus en plus sophistiqués, le volume de données qui résulte d'une acquisition et qui doit être mémorisé à bord et ensuite vidé vers le sol est de moins en moins prédictible. Il dépend des données acquises. Par exemple, dans le cas d'instruments optiques, la présence de nuages sur la zone observée permet une forte compression dont le résultat est un faible volume de don-

nées à mémoriser et à vider.

Dans de telles conditions, la façon classique de gérer ces satellites devient de plus en plus problématique. En effet, jusqu'à maintenant, toutes les décisions sont prises hors-ligne au sol et le satellite est un simple exécutant qui ne prend, ni ne modifie aucune décision. Typiquement, chaque jour, un plan d'activité incluant les activités d'acquisition et de vidage avec des dates de début précises est construit au sol pour le jour suivant. Ce plan est téléchargé vers le satellite via une station de contrôle et ensuite exécuté tel quel sans aucune modification. Dans un contexte où le volume de données généré par les acquisitions est incertain, si les volumes maximaux sont considérés au moment de la planification, les plans seront toujours exécutables, mais le système pourra être sous-utilisé. Si des volumes espérés ou n'importe quels volumes inférieurs aux volumes maximaux sont considérés, les plans pourront ne pas être exécutables.

Une option alternative consiste à modifier au moins en partie la façon de gérer ces satellites, en prenant les décisions de vidage le plus tard possible, quand les acquisitions sont réalisées et les volumes générés sont connus. Comme ces satellites ne sont pas de façon permanente accessibles par une station de contrôle et comme les volumes générés sont d'abord connus à bord, ces décisions doivent et peuvent être prises à bord. Par exemple, juste avant une fenêtre de visibilité d'une station de réception pendant laquelle des vidages sont possibles, le satellite peut construire un plan de vidage qui prenne en compte les volumes générés par toutes les acquisitions déjà réalisées.

Dans cette étude, nous considérons le contexte des futurs satellites Post-Pléiades : la génération des satellites d'observation optique de la Terre qui suivra celle des satellites Pléiades actuellement opérationnels. Dans un tel contexte, de nouvelles contraintes de vidage doivent être prises en compte telles que la mémorisation des données sur différentes tranches mémoire, l'utilisation de plusieurs canaux de vidage ou l'emploi de clés de cryptage. De plus, les critères de vidage à optimiser sont le nombre et l'importance des acquisitions vidées, le délai entre acquisitions et vidages et le partage équitable de la ressource satellite entre utilisateurs.

Le premier objectif de l'étude était de développer au dessus d'une bibliothèque générique d'optimisation sous contraintes des algorithmes efficaces capables de prendre des décisions de vidage de façon autonome à bord. Le second objectif était d'évaluer l'impact opérationnel de l'utilisation de tels algorithmes par rapport à la façon classique de gérer ces satellites entièrement depuis le sol.

Le papier est organisé de la façon suivante. Dans la section suivante, l'architecture mixte de décision bord-sol que nous supposons est décrite. Puis, le problème de planification des vidages de données est informellement décrit et un modèle à base de contraintes de ce problème est proposé et analysé. L'approche algorithmique retenue, reprenant les idées de recherche locale à base de contraintes (*Constraint-based Local Search*, CLS [4]) et utilisant la bibliothèque générique InCELL (*Invariant-based Constraint EvaLuation Library* [10]), est ensuite décrite. Après une présentation des divers scénarios utilisés pour l'évaluation, les résultats expérimentaux permettent de comparer l'efficacité des différentes variantes algorithmiques et d'évaluer l'impact positif de la prise de décision à bord.

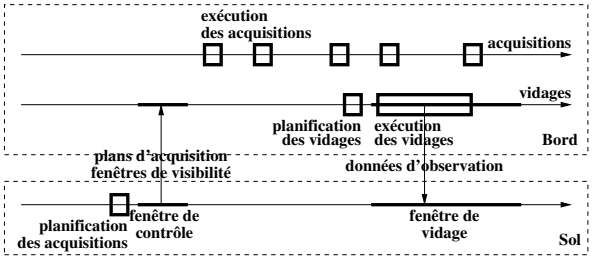


FIGURE 1 – Architecture de prise de décision.

2 Architecture de prise de décision

Nous considérons le contexte des futurs satellites Post-Pléiades avec les hypothèses suivantes sur le système physique :

- l'instrument d'observation est fixe sur le satellite, mais l'antenne de vidage des données est mobile avec une vitesse et un angle de débattement maximaux ;
- pour observer une zone sol, l'instrument d'observation et donc tout le satellite doit être pointé vers elle ; pour vider des données vers une station sol, l'antenne de vidage doit être pointée vers elle ;
- grâce à des actionneurs gyroscopiques, le satellite est agile et capable de se mouvoir rapidement en attitude selon les trois axes autour de son centre de gravité tout en se déplaçant sur son orbite ;
- acquisitions et vidages peuvent être réalisés en parallèle.

Dans ce contexte, la figure 1 montre l'architecture de prise de décision que nous supposons.

Planification des acquisitions Les plans d'acquisition sont construits comme d'habitude hors-ligne au sol. Ceci est justifié par le fait que leur construction est coûteuse en temps de calcul et que les requêtes d'acquisition sont d'abord connues au sol. Ces plans ne considèrent ni les limitations en termes de mémoire et de vidage, ni les activités de vidage. D'un plan d'acquisition, il est possible de déduire, pour chaque station de réception st, les fenêtres sur lesquelles vider des données vers st est effectivement possible (pointer l'antenne mobile vers st est possible, en prenant en compte la position et l'orientation du satellite, la

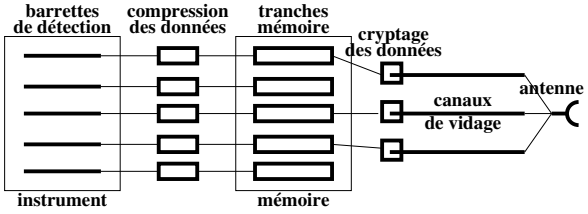


FIGURE 2 – Comment les données sont produites, mémorisées et vidées.

position de la station et l'angle maximum de débattement de l'antenne). Le plan d'acquisition et les fenêtres de visibilité résultantes sont alors téléchargés vers le satellite via n'importe quelle station de contrôle.

Exécution des acquisitions Les plans d'acquisition sont exécutés à bord sans aucune modification, sauf quand une acquisition a conduirait à un dépassement mémoire. Dans ce cas, a est réalisée si et seulement si il est possible de libérer suffisamment de mémoire en effaçant des acquisitions de plus faible priorité.

Planification des vidages Les plans de vidage sont construits à bord avant une fenêtre de visibilité ou un groupe de fenêtres chevauchantes. Ceci est justifié par le fait que les volumes générés par les acquisitions sont d'abord connus à bord. Ces plans prennent en compte les volumes exacts pour les acquisitions déjà réalisées et les volumes maximaux pour les acquisitions qui se terminent durant la(les) fenêtre(s) de visibilité concernées.

Exécution des vidages Les plans de vidage sont ensuite exécutés de façon flexible, en prenant en compte le fait que les volumes générés par les acquisitions qui se terminent durant la(les) fenêtre(s) de visibilité peuvent être plus faibles que les volumes maximaux pris en compte par la planification. Cela peut permettre de commencer des vidages plus tôt que ce qui avait été planifié.

3 Problème de planification des vidages

La figure 2 montre comment les données sont produites, mémorisées et vidées.

Production et mémorisation des données Pour obtenir une observation multi-fréquence, chaque acquisition active un sous-ensemble de barrettes de détection. Elle produit un ensemble de fichiers de données, chacun étant mémorisé sur une tranche mémoire avec une taille qui dépend du taux de compression effectif.

Vidage des données Le vidage peut utiliser plusieurs canaux d'émission concurrents. Chaque fichier doit être vidé sans interruption sur un canal. Le débit de vidage est une fonction constante par morceaux de la distance satellite-station. Du fait du mouvement du satellite sur son orbite et de la Terre sur elle-même, la distance satellite-station évolue. En conséquence la durée de vidage d'un fichier dépend de l'instant auquel il débute. Les fichiers issus d'une acquisition peuvent être vidés dans n'importe quel ordre sur n'importe quels canaux, mais doivent être tous vidés dans la même fenêtre de visibilité. Canaux et tranches mémoire sont des ressources non partageables. Cela implique que, si deux fichiers sont mémorisés sur la même tranche ou utilisent le même canal, leurs vidages ne peuvent pas se chevaucher.

Cryptage des données Les données sont cryptées avant vidage. Une clé est associée à chaque utilisateur. Physiquement, un composant de cryptage est associé à chaque canal, ce qui permet à des données associées à des utilisateurs différents d'être vidées en parallèle. Une table de changement de clé, qui contient tous les changements à effectuer et leurs instants précis, est associée à chaque canal, ce qui permet à des données associées à des utilisateurs différents d'être vidées en séquence sur chaque canal. Le nombre de changements qu'il est possible de mémoriser dans la table est limité. Réinitialiser cette table prend un certain temps et interrompt les vidages sur tous les canaux.

Fenêtres de vidage Vider des données est possible uniquement à l'intérieur des fenêtres de visibilité. De plus, en fonction de l'utilisateur, seules certaines stations et donc certaines fenêtres sont autorisées. Nous supposons qu'une fenêtre de visibilité peut inclure plusieurs fenêtres de vidage, chacune avec sa table de changement de clé associée. En conséquence, entre deux fenêtres de vidage successives, un temps minimum est nécessaire pour réinitialiser la table. De plus, si les deux fenêtres successives sont associées à deux stations différentes, passer de l'une à l'autre prend un certain temps requis pour pointer l'antenne vers la nouvelle station. Ce temps de transition dépend de l'instant auquel la transition est enclenchée du fait du mouvement du satellite sur son orbite et sur lui-même et du mouvement de la Terre sur elle-même.

Dates de vidage au plus tôt et au plus tard Le vidage d'une acquisition doit débuter après l'acquisition elle-même et se terminer avant une date limite au delà de laquelle les données n'ont plus aucune valeur.

La figure 3 représente un plan de vidage valide où les décisions d'affectation et de vidage ont été prises, mais les dates de début de vidage n'ont pas encore été fixées (plan temporellement flexible). Ce plan implique 4 acquisitions A, B, C, and D, chacune produisant 5 fichiers (A1, A2, A3, A4 et A5 pour l'acquisition A) distribués sur 5 tranches mémoire, 2 fenêtres de visibilité chevauchantes (Vw1 vers la station S1 et Vw2 vers la station S2), une fenêtre de vidage par fenêtre de visibilité et 3 canaux de vidage.

Seules les contraintes temporelles de précédence sont représentées : (1) séquences de vidage sur chaque canal, (2)

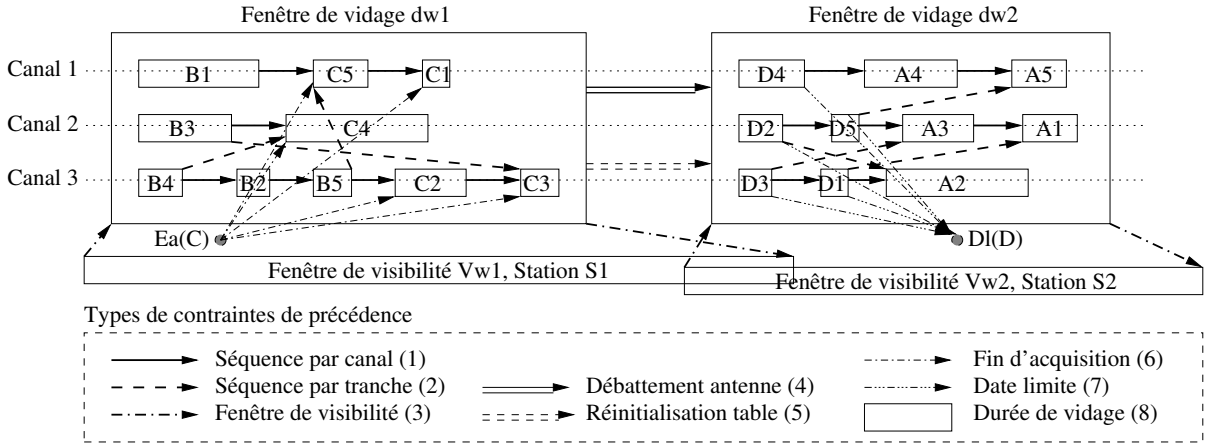


FIGURE 3 – Exemple de plan de vidage avec ses contraintes temporelles de précédence.

séquences de vidage depuis chaque tranche, (3) vidages dans les fenêtres de visibilité, temps de transition entre deux fenêtres de vidage successives dû (4) au mouvement de l'antenne et (5) à la ré-initialisation de la table de changement de clé, (6) acquisition avant vidage (pour l'acquisition C qui se termine à l'instant Ea(C) dans la fenêtre Vw1) et (7) vidage avant la date limite (pour l'acquisition D dont la date limite de vidage Dl(D) se situe dans la fenêtre Ww2). Les durées de vidage (8) sont implicitement représentées par les rectangles associés à chaque fichier.

Les contraintes non temporelles ne sont pas représentées : (9) tous les fichiers issus d'une acquisition vidés dans une même fenêtre de visibilité dont la station associée est autorisée et (10) nombre maximum de changements de clé sur chaque canal pour chaque fenêtre de vidage.

4 Modèle

4.1 Variables de décision

Les variables de décision peuvent être classées en variables non temporelles et variables temporelles. Les variables non temporelles incluent :

- un nombre ndw de fenêtres de vidage et la séquence $dwSeq$ (de longueur ndw) de fenêtres de vidage ;
- pour chaque acquisition a, la fenêtre de vidage dw_a à l'intérieur de laquelle a est vidée et, pour chaque fichier associé f, le canal $ch_{a,f}$ sur lequel f est vidé ;
- pour chaque fenêtre de vidage w, la fenêtre de visibilité associée vw_w, pour chaque canal c, la séquence $cSeq_{w,c}$ de vidages de fichiers sur c et, pour chaque tranche m, la séquence $mSeq_{w,m}$ de vidages de fichiers depuis m.

Les variables temporelles incluent :

- pour chaque acquisition a et chaque fichier associé f, les dates de début et de fin $sdf_{a,f}$ et $edf_{a,f}$ du vidage de f ;
- pour chaque fenêtre de vidage w, ses dates de début et de fin sdw_w et edw_w.

4.2 Contraintes

Sur la base de cette définition des variables de décision, toutes les contraintes de (1) à (10) informellement présentées à la fin de la section 3 peuvent être formulées.

Beaucoup d'entre elles sont des contraintes temporelles simples [2] comme la contrainte (3) qui impose que toute fenêtre de vidage soit incluse dans sa fenêtre de visibilité associée, avec $\mathbf{Sw}_{vw}$ and $\mathbf{Ew}_{vw}$ les dates de début et de fin d'une fenêtre de visibilité vw :

$$\forall w \in [\![1; ndw]\!] : \ (\mathbf{Sw}_{vw_w} \leq sdw_w) \wedge (edw_w \leq \mathbf{Ew}_{vw_w})$$

ou les contraintes (6) et (7) qui expriment que le vidage d'une acquisition doit commencer après l'acquisition elle-même et finir avant la date limite, avec **Na** le nombre d'acquisitions, $\mathbf{Nf}_a$ le nombre de fichiers résultant d'une acquisition a, $\mathbf{Ea}_a$ la date de fin d'une acquisition a et $\mathbf{Dl}_a$ sa date limite de vidage :

$$\forall a \in [\![1; \mathbf{Na}]\!], \forall f \in [\![1; \mathbf{Nf}_a]\!] :$$
$$(\mathbf{Ea}_a \leq sdf_{a,f}) \wedge (edf_{a,f} \leq \mathbf{Dl}_a)$$

Certaines d'entre elles sont des contraintes temporelles simples dépendantes du temps [11] comme la contrainte (8) qui exprime la durée du vidage d'un fichier en fonction de la date de début de vidage, avec $\mathbf{V}_{a,f}$ le volume d'un fichier f issu d'une acquisition a et **DuDl** la durée de vidage en fonction du volume à vider, de la fenêtre de visibilité et de la date de début de vidage dans cette fenêtre :

$$\forall a \in [\![1; \mathbf{Na}]\!], \forall f \in [\![1; \mathbf{Nf}_a]\!] :$$
$$edf_{a,f} - sdf_{a,f} = \mathbf{DuDl}(\mathbf{V}_{a,f}, vw_{dw_a}, sdf_{a,f})$$

Certaines d'entre elles sont des contraintes non temporelles comme la contrainte (8) qui impose qu'une acquisition soit vidée vers une station autorisée, avec $\mathbf{St}_{vw}$ la station associée à une fenêtre de visibilité vw, $\mathbf{Ss}_u$ l'ensemble des stations autorisées par un utilisateur u et $\mathbf{U}_a$ l'utilisateur demandeur d'une acquisition a :

$$\forall a \in [\![1; \mathbf{Na}]\!] : \ \mathbf{St}_{vw_{dw_a}} \in \mathbf{Ss}_{\mathbf{U}_a}$$

4.3 Critère d'optimisation

Dans un but de simplification, le critère d'optimisation considéré dans ce papier est purement utilitariste et ne considère pas l'objectif de partage équitable de l'usage du satellite entre ses différents utilisateurs. Il est défini comme un vecteur d'utilités, avec une utilité par niveau de priorité. Pour chaque niveau de priorité p, l'utilité U_p est simplement définie comme la somme des poids des acquisitions vidées de priorité p, pondérés par leur coefficient de fraîcheur, de façon à favoriser des délais courts entre acquisition et vidage :

$$\forall p \in [\![1; \mathbf{Np}]\!] : \ U_p = \sum_{a \in [\![1; \mathbf{Na}]\!] \mid \mathbf{P}_a = p} \mathbf{W}_a \cdot \mathbf{Fr}_a(eda_a)$$

où **Np** est le nombre de niveaux de priorité, **Na** le nombre d'acquisitions, $\mathbf{P}_a$ et $\mathbf{W}_a$ la priorité et le poids d'une acquisition a, $\mathbf{Fr}_a$ son niveau de fraîcheur (compris entre 0 et 1) en fonction (monotone décroissante) de sa date eda_a de fin de vidage : $eda_a = \max_{f \in [\![1; \mathbf{Nf}_a]\!]} edf_{a,f}$.

Deux plans de vidage sont comparés en comparant leurs deux vecteurs d'utilité de façon lexicographique des priorités 1 à **Np** : n'importe quelle amélioration à un niveau de priorité p est préféré à n'importe quelle amélioration à des niveaux de priorité inférieurs à p.

5 Analyse du problème

Toujours dans un but de simplification, nous supposons dans ce papier qu'au plus une fenêtre de vidage (éventuellement aucune) est associée à une fenêtre de visibilité station et que les fenêtres de vidage résultantes sont ordonnées suivant la date de début de leur fenêtre de visibilité associée. Voir l'exemple de la figure 3 où nous considérons deux fenêtres de vidage $dw1$ et $dw2$ et la séquence $[dw1, dw2]$.

Dans ces conditions, le problème à traiter combine trois sous-problèmes connectés :

1. un problème d'affectation où une fenêtre de vidage (éventuellement aucune) est associée à chaque acquisition : variables dw_a ;
2. un problème d'ordonnancement où un canal (éventuellement aucun) est associé à chaque fichier et où les vidages de fichiers sont ordonnés sur chaque canal et chaque tranche : variables $ch_{a,f}$, $cSeq_{w,c}$ et $mSeq_{w,m}$;
3. un problème temporel où les dates de début et de fin sont fixées : variables $sdf_{a,f}$, $edf_{a,f}$, sdw_w et edw_w.

Problème d'affectation Le problème d'affectation est proche du problème *Multi-knapsack* [6] où les objets sont les acquisitions et les sacs sont les fenêtres, mais les premiers sacs sont préférés et des conflits peuvent exister entre sacs (fenêtres chevauchantes).

Problème d'ordonnancement Dans chaque fenêtre de vidage, le problème d'ordonnancement est proche du problème *Flexible Open-shop Scheduling* [8] avec deux types de ressources non partageables : canaux et tranches. Chaque vidage requiert une ressource de chaque type, mais le choix du canal est libre alors que la tranche est préaffectée. L'ordre des vidages est libre.

Problème temporel Quand les problèmes d'affectation et d'ordonnancement ont été résolus, c'est-à-dire quand toutes les variables non temporelles ont été affectées, le problème temporel résultant a la forme d'un *Simple Temporal Network* (STN [2]). Les seules exceptions sont les contraintes de durée de vidage et de transition entre fenêtres de vidage qui sont dépendantes du temps (durées de vidage et de transition entre fenêtres dépendent des instants auxquels vidages et transitions commencent). Le résultat est un *Time-dependent* STN (TSTN [11]) pour lequel les techniques STN peuvent être étendues et des algorithmes polynomiaux peuvent décider de la cohérence et, en cas de cohérence, calculer les dates au plus tôt et au plus tard de toutes les variables temporelles.

6 Algorithmes de planification

Globalement, les algorithmes que nous avons développés sont des algorithmes gloutons heuristiques non chronologiques qui, de façon séquentielle et sans aucun *backtrack*, choisissent une acquisition et tentent de l'ajouter au plan de vidage. Ce choix algorithmique est justifié par les limitations bord en termes de temps et de ressources de calcul. Il faut cependant souligner que ce choix n'est pas définitif et que d'autres choix sont possibles sur la base du même modèle, tels que des algorithmes de type *squeaky wheel optimization* [5], *tabu search* [3] ou *simulated annealing* [7]. Il faut aussi souligner que ces algorithmes gloutons incluent des mécanismes sophistiqués de choix

d'affectation et d'ordonnancement des vidages, de vérification des contraintes non temporelles et de propagation des contraintes temporelles. Conformément à l'analyse du problème en section 5, ces algorithmes comportent trois parties : affectation des vidages, ordonnancement des vidages et vérification et propagation des contraintes.

6.1 Affectation des vidages

Les algorithmes d'affectation prennent en entrée un plan de vidage courant cohérent et un ensemble d'acquisitions candidates (non encore affectées). Ils fournissent en sortie une acquisition a (sélectionnée parmi les candidates) et une fenêtre de vidage pour a. Nous avons développé deux algorithmes d'affectation : **MaxWeight** et **MinRegret**.

MaxWeight est l'algorithme le plus basique. À chaque étape, il sélectionne une acquisition a de priorité maximum et de poids maximum, avec tirage aléatoire en cas d'égalité, et sélectionne la première fenêtre (au sens chronologique) dans laquelle insérer a est possible.

MinRegret est un peu plus sophistiqué. Il maintient pour chaque acquisition a la première et la seconde fenêtre ($w1_a$ et $w2_a$) dans lesquelles insérer a est possible, ainsi que le regret qui résulterait du fait de ne pas choisir la première ($w1_a$). Si $eda1_a$ et $eda2_a$ sont les dates de fin de vidage de a qui résulterait du choix de $w1_a$ et de $w2_a$, le regret R peut être défini comme suit :

$$\forall a \in [\![1; \mathbf{Na}]\!] : \; R(a) = \mathbf{W}_a \cdot (\mathbf{Fr}_a(eda1_a) - \mathbf{Fr}_a(eda2_a))$$

En s'inspirant des heuristiques classiques utilisées pour traiter les problèmes de sac-à-dos, l'algorithme sélectionne à chaque étape une acquisition a de priorité maximum et de ratio maximum $R(a)/\sum_{f \in [\![1;\mathbf{Nf}_a]\!]} \mathbf{V}_{a,f}$ entre regret et volume et sélectionne la première fenêtre ($w1_a$) pour a.

6.2 Ordonnancement des vidages

Les algorithmes d'ordonnancement prennent en entrée un plan de vidage courant cohérent, une fenêtre de vidage w et une acquisition a à insérer dans w. Ils produisent un nouveau plan de vidage cohérent avec a inséré dans w ou un échec si l'insertion a échoué.

Nous avons développé trois algorithmes d'ordonnancement : **EnQueue**, **IdleFill** et **Scheduler**. Les deux premiers sont des algorithmes gloutons, tandis que le troisième utilise des mécanismes de recherche locale. De plus, les deux premiers produisent des ordonnancements où les vidages des acquisitions sont totalement ordonnés : vider $a1$ avant $a2$ implique que, sur chaque canal et chaque tranche, aucun fichier de $a2$ n'est vidé avant un fichier de $a1$. Au contraire, le troisième algorithme construit des ordonnancements où il est possible d'entrelacer les vidages des acquisitions.

EnQueue est l'algorithme le plus basique. Pour insérer une acquisition a dans une fenêtre w, il l'insère systématiquement à la fin de la séquence courante de vidage.

IdleFill remédie au défaut le plus évident de l'algorithme précédent : comme certaines acquisitions se terminent au cours de fenêtres de visibilité, certains vidages doivent attendre la fin d'acquisition ; ceci peut engendrer des périodes d'inactivité dans l'ordonnancement. Pour éviter ce phénomène, **IdleFill** insère une acquisition a à la première position dans la séquence courante de vidage où au moins un canal est inactif et l'insertion est possible.

Pour les deux algorithmes, suivant une heuristique de type plus contraint en premier, l'insertion de tous les fichiers de a est réalisée de façon séquentielle suivant un ordre de volume décroissant. Toujours, pour limiter les périodes d'inactivité dans l'ordonnancement, pour chaque fichier f, s'il existe un canal c sur lequel le dernier fichier f' vient de la même tranche que f, alors f est placé sur c juste après f' ; sinon f est placé sur un canal c qui permet un vidage au plus tôt.

Scheduler est un algorithme de recherche locale inspiré de mécanismes efficaces utilisés pour traiter des problèmes classiques d'ordonnancement. Il est appelé en cas d'échec des algorithmes gloutons précédents et peut démarrer de l'ordonnancement qu'ils produisent. Dans cet ordonnancement, les contraintes sur les fenêtres de visibilité sont violées et le but est de produire un ordonnancement plus court qui les satisfasse. Pour cela, l'algorithme réalise une séquence de mouvements locaux dont le but est de réduire la longueur du chemin critique (la séquence de vidages dont la longueur induit la longueur de l'ordonnancement ; si rien n'est changé dans cette séquence, la longueur de l'ordonnancement ne sera pas réduite et les contraintes sur les fenêtres de visibilité resteront violées). Ces mouvements locaux sont conçus pour éviter de produire des cycles dans l'ordonnancement : ordre sur les canaux contradictoire avec l'ordre sur les tranches.

6.3 Vérification et propagation des contraintes

La librairie générique InCELL (*Invariant-based Constraint EvaLuation Library* [10]) est utilisée pour modéliser variables, contraintes et critère, pour vérifier les contraintes non temporelles, pour propager les contraintes temporelles et pour évaluer le critère. InCELL tire son inspiration des idées de *Constraint-based Local Search* (CLS [4]).

Avec CLS, comme avec d'autres approches déclaratives pour l'optimisation sous contraintes, l'utilisateur définit un modèle de son problème en termes de variables de décision, de contraintes et de critère à optimiser. Puis, il définit un algorithme de recherche gloutonne ou locale [1] dans l'espace d'affectation des variables. Le modèle n'est pas utilisé pour propager les contraintes comme avec les al-

gorithmes classiques de recherche arborescente, mais pour vérifier les contraintes et évaluer le critère en fonction de l'affectation courante. Parce que le nombre de mouvements locaux réalisés en un temps limité est une des clés du succès de ces algorithmes, des techniques efficaces sont utilisées pour réaliser chaque mouvement local aussi rapidement que possible. Ces techniques utilisent une traduction du modèle (variables, contraintes et critère) en un DAG (*Directed Acyclic Graph*) d'invariants. Chaque invariant a un ensemble d'entrées et une sortie. Il maintient une certaine fonction des entrées vers la sortie, par exemple le fait qu'une variable est égale à la somme d'autres variables : $x = \sum_{i=1}^{N} y_i$ et il la maintient de façon incrémentale : sur cet exemple, si la valeur d'une certaine variable y_k est modifiée, il n'est pas nécessaire de recalculer toute la somme ; il suffit d'ajouter à x la différence entre la nouvelle et l'ancienne valeur de y_k. Globalement, après chaque changement dans l'affectation des variables, le DAG d'invariants est ré-évalué de façon paresseuse et incrémentale : seuls les invariants dont une des entrées est modifiée sont ré-évalués de façon incrémentale.

InCELL est une implémentation des idées de CLS visant plus particulièrement les problèmes d'ordonnancement et donc la gestion du temps et des ressources. Avec InCELL, des invariants à entrées et sorties multiples permettent de représenter des expressions, des contraintes arithmétiques et logiques et des contraintes sur le temps et les ressources. De plus, dans InCELL, les contraintes temporelles simples [2] de la forme $y - x \leq D$, où x et y représentent deux positions temporelles et D une constante, sont gérées de façon spécifique : les variables temporelles ne sont pas affectées comme les variables non temporelles le sont ; les techniques de type STN sont utilisées pour propager les contraintes temporelles, pour vérifier leur cohérence et pour calculer les dates au plus tôt et au plus tard de chaque variable temporelle. Ce traitement spécifique est justifié par l'existence d'algorithmes polynomiaux capables de vérifier la cohérence d'un STN et de calculer les dates au plus tôt et au plus tard de chaque variable du STN. De tels algorithmes n'existent en général pas pour les contraintes non temporelles. En plus, InCELL permet d'exprimer et de traiter de la même façon des contraintes temporelles simples dépendantes du temps [11] de la forme $y - x \leq D(x, y)$, où D n'est plus une constante, mais une fonction de x et y.

Une fois que les contraintes non temporelles ont été vérifiées et les contraintes temporelles propagées par InCELL, le résultat est un plan de vidage qui affecte à chaque variable temporelle sa date au plus tôt, car il est préférable de vider les données au plus tôt. A partir de ce plan de vidage, les tables de changement de clé sont fixées avec des dates de changement précises pour chaque fenêtre de vidage et chaque canal.

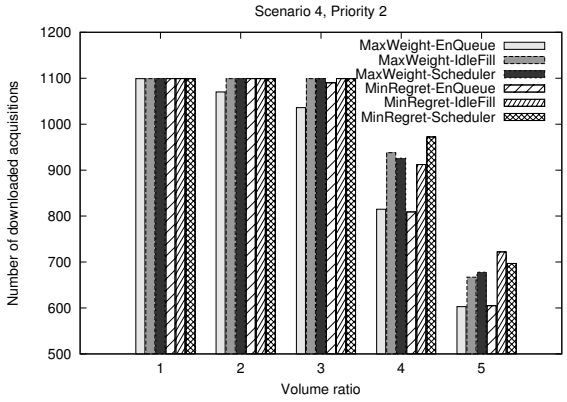


FIGURE 4 – Nombre d'acquisitions vidées de priorité 2 sur le scénario 4. On cherche à maximiser ce nombre.

	Scénario 1		Scénario 4	
	moyen	max	moyen	max
MaxWeight-EnQueue	0.041	0.221	0.048	0.460
MaxWeight-IdleFill	0.035	0.221	0.049	0.686
MaxWeight-Scheduler	0.232	4.149	0.655	21.708
MinRegret-EnQueue	0.161	1.434	0.239	3.531
MinRegret-IdleFill	0.152	1.544	0.215	2.839
MinRegret-Scheduler	0.157	1.521	2.116	78.511

TABLE 1 – Temps de calcul moyen et maximum (en secondes) sur les scénarios 1 et 4 (processeur Intel i5-520, 1.2GHz, 4GBRAM).

7 Algorithmes d'exécution

Dans l'architecture de prise de décision retenue, des plans de vidage sont construits avant chaque fenêtre ou groupe de fenêtres de visibilité en faisant l'hypothèse de volumes maximaux pour toutes les acquisitions se terminant pendant la(les) fenêtre(s) de visibilité. Si les volumes réels sont plus faibles que les volumes maximums, il est possible de démarrer des vidages plus tôt que prévu et donc d'améliorer la qualité du plan sans modifier les décisions d'affectation et d'ordonnancement.

Pour implémenter une exécution flexible et réactive des plans, nous nous inspirons de l'approche *Partial Order Schedule* (POS [9]) et construisons à partir de n'importe quel plan de vidage un graphe de précédence (un DAG) qui représente toutes les précédences qui doivent être satisfaites par l'exécution. Une fois ce graphe construit, il peut être exécuté dans un ordre topologique : chaque nœud est exécuté dès que tous ses prédécesseurs dans le graphe ont été exécutés. On peut facilement montrer que, du fait des volumes maximaux pris en compte par la planification, une telle exécution ne conduira jamais à une violation des

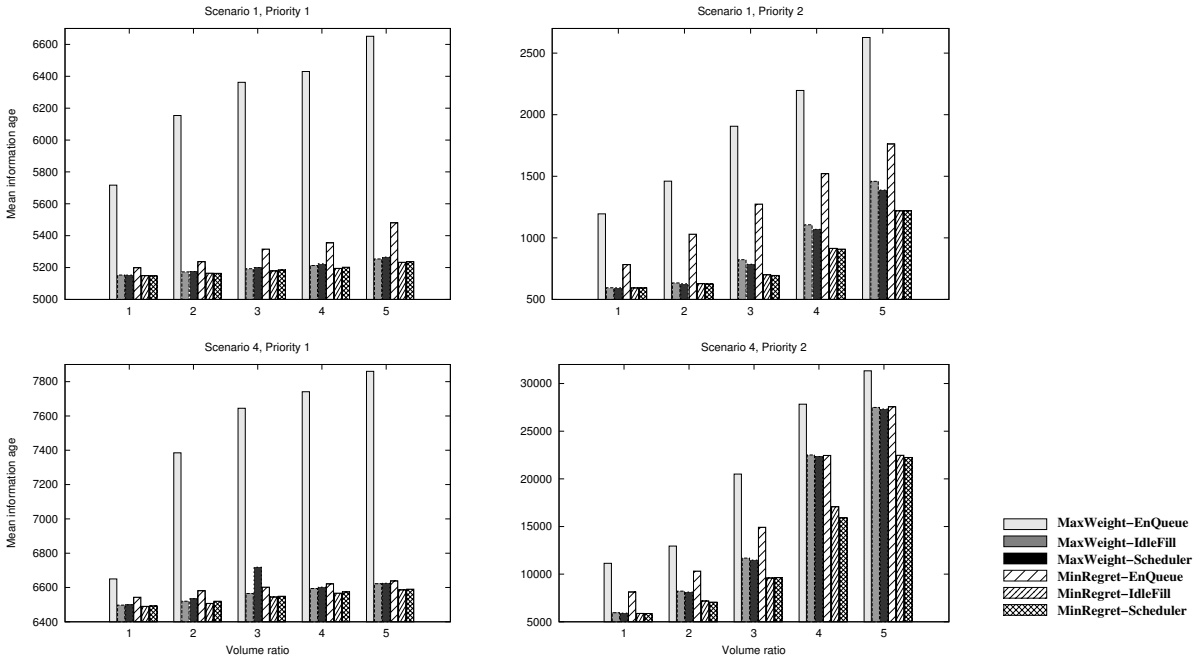


FIGURE 5 – Age moyen de l'information (en secondes) pour les priorités 1 et 2 (à gauche et à droite) sur les scénarios 1 et 4 (en haut et en bas). On cherche à minimiser cet âge.

contraintes sur les fenêtres de visibilité et les dates limites de vidage.

8 Résultats expérimentaux

8.1 Scénarios

L'architecture de prise de décision et les divers algorithmes de planification ont été expérimentés sur différents scénarios réalistes fournis par Airbus Defence and Space. Chaque scénario couvre un jour d'activité du satellite.

Ces scénarios impliquent 5 tranches mémoire, 3 canaux de vidage, 5 utilisateurs, 2 niveaux de priorité, de 3 à 23 stations de réception, de 20 à 115 fenêtres de visibilité associées et 1364 acquisitions à vider. Comme chaque acquisition produit 5 fichiers, le nombre de fichiers à vider est égal à 6820. Si $Vmax$ est le volume maximum d'un fichier (sans compression), son volume réel (après compression) est généré de façon aléatoire entre $Vmax/4$ et $Vmax$. Les scénarios diffèrent suivant le nombre de stations de réception disponibles et la façon dont les acquisitions sont distribuées entre utilisateurs et niveaux de priorité.

Nous présentons les résultats obtenus sur deux scénarios typiques : le scénario 1 qui implique un grand nombre de stations de réception (23) induisant de nombreuses opportunités de vidage et le scénario 4 qui implique un petit nombre de stations de réception (3) induisant peu d'opportunités de vidage. De plus, nous présentons les résultats obtenus sur les deux scénarios avec différentes hypothèses concernant le volume de données généré par une acquisition avant compression : ratio en volume variant de 1 à 5 conduisant à des problèmes de plus en plus sur-contraints. Par exemple, le scénario 1 avec un ratio en volume égal à 1 est largement sous-contraint : chaque acquisition peut être vidée dans la première fenêtre de visibilité suivant l'acquisition. Au contraire, le scénario 4 avec un ratio en volume égal à 5 est fortement sur-contraint : les acquisitions ne peuvent pas toutes être vidées et un grand nombre reste en mémoire à bord à la fin de la journée

8.2 Comparaison ente algorithmes de planification

Nous comparons tout d'abord les six algorithmes de planification résultant de la combinaison des algorithmes d'affectation et d'ordonnancement : **MaxWeight** ou **MinRegret** pour l'affectation et **EnQueue**, **IdleFill** ou **Scheduler** pour l'ordonnancement.

En ce qui concerne le nombre d'acquisitions vidées, toutes les acquisitions sont vidées dans le scénario 1 et toutes les acquisitions de priorité 1 le sont dans le scénario 4, mais pas toutes les acquisitions de priorité 2 avec des ratios en volume élevés. La figure 4 montre le nombre d'acquisitions de priorité 2 vidées dans le scénario 4 en fonction

Scénario 1 : Nombre d'acquisitions vidées						
ratio en volume		1	2	3	4	5
prio. 1	sol	269	269	269	269	269
	bord	269	269	269	269	269
prio. 2	sol	1087	1087	1087	988	729
	bord	1087	1087	1087	1087	1087

Scénario 1 : Age moyen de l'information						
ratio en volume		1	2	3	4	5
prio. 1	sol	5167	5200	5232	5268	5419
	bord	5149	5165	5179	5195	5233
prio. 2	sol	610	692	993	1505	2403
	bord	593	628	700	915	1219

Scénario 4 : Nombre d'acquisitions vidées						
ratio en volume		1	2	3	4	5
prio. 1	sol	244	244	244	244	244
	bord	244	244	244	244	244
prio. 2	sol	1099	1099	595	297	246
	bord	1099	1099	1099	912	722

Scénario 4 : Age moyen de l'information						
ratio en volume		1	2	3	4	5
prio. 1	sol	6639	6725	6758	6914	7125
	bord	6490	6507	6544	6567	6586
prio. 2	sol	6700	12898	13539	13771	11784
	bord	5875	7180	9591	17073	22474

TABLE 2 – Comparaison entre planification sol et bord : nombre d'acquisitions vidées (à gauche) et âge moyen de l'information (en secondes ; à droite) pour les priorités 1 et 2 sur les scénarios 1 et 4 (en haut et en bas).

du ratio en volume (de 1 à 5).

La figure 5 montre l'âge moyen de l'information en secondes (distance moyenne entre acquisition et vidage sur toutes les acquisitions vidées) pour les priorités 1 et 2 (à gauche et à droite) dans les scénarios 1 et 4 (en haut et en bas) en fonction du ratio en volume (de 1 à 5). Le fait que l'âge de l'information pour les acquisitions de priorité 1 soit plus grand que pour celles de priorité 2 est dû à un plus petit nombre de stations de réception autorisées pour les acquisitions de priorité 1.

Ces résultats en termes de nombre d'acquisitions vidées et d'âge moyen de l'information montrent que, pour l'affectation, **MinRegret** est meilleur que **MaxWeight** et que, pour l'ordonnancement, **IdleFill** est clairement meilleur que **EnQueue**, mais que **Scheduler** est seulement légèrement meilleur que **IdleFill**. Finalement, en termes de qualité des plans, **MinRegret-IdleFill** et **MinRegret-Scheduler** semblent être les meilleurs algorithmes.

Le tableau 1 montre le temps de calcul moyen et maximum en secondes (sur tous les appels à la planification, avant chaque fenêtre ou groupe de fenêtres de visibilité) dans les scénarios 1 et 4, uniquement pour un ratio en volume de 1 : ces temps de calcul ne changent pas de façon significative pour des ratios supérieurs (de 2 à 5).

Ces résultats en termes de temps de calcul montrent que **MaxWeight-EnQueue** and **MaxWeight-IdleFill** sont les plus efficaces. Remplacer **MaxWeight** par **MinRegret** multiplie le temps de calcul environ par 5. Remplacer **EnQueue** ou **IdleFill** par **Scheduler** peut le multiplier par 30.

En conséquence, le choix du meilleur algorithme dépend des ressources de calcul disponibles à bord. Si elles sont suffisantes, **MinRegret-IdleFill** semble être le meilleur candidat. Sinon, il peut être remplacé par **MaxWeight-IdleFill**.

8.3 Comparaison entre planification au sol ou à bord

Nous avons ensuite comparé ce qui peut être obtenu en planifiant les vidages au sol comme cela est actuellement fait (planification sur un horizon d'un jour, faisant l'hypothèse de volumes maximaux, fixant des dates précises de vidage et ne permettant aucune flexibilité à bord) et ce qui peut être obtenu en planifiant les vidages à bord (planification des vidages avant chaque fenêtre ou groupe de fenêtres de visibilité, suivie d'une exécution flexible). Dans les deux cas, l'algorithme de planification utilisé est **MinRegret-IdleFill**.

Le tableau 2 montre le nombre d'acquisitions vidées (à gauche) et l'âge moyen de l'information (à droite) pour les priorités 1 et 2 dans les scénarios 1 et 4 (en haut et en bas) en fonction du ratio en volume (de 1 à 5).

Ces résultats montrent l'impact positif de la planification à bord en termes de nombre d'acquisitions vidées qui peut être multiplié par 3 pour les acquisitions de priorité 2 à partir de ratios en volume de 3 ou 4, mais aussi en termes d'âge moyen de l'information qui peut être divisé par 2 pour les acquisitions de priorité 2. Dans certains cas, nous pouvons cependant observer que la planification à bord augmente l'âge moyen de l'information. Ceci est dû à un plus grand nombre d'acquisitions vidées, mais vidées assez tard.

En plus, ces résultats montrent le bon comportement des algorithmes de planification bord qui privilégient les acquisitions de priorité 1 : quels que soient le scénario et le ratio en volume, les acquisitions de priorité 1 sont toutes vidées et, quand le ratio en volume augmente (problème de plus en plus sur-contraint), l'âge moyen de l'information croit moins vite pour les acquisitions de priorité 1 que pour celles de priorité 2.

9 Conclusion

Dans ce papier, nous avons montré comment l'incertitude sur le volume de données généré par les acquisitions peut être gérée en utilisant des algorithmes bord simples et efficaces de planification et d'exécution des vidages qui combinent une recherche gloutonne ou locale, un modèle à base de contraintes et des appels à une librairie générique d'évaluation et de propagation des contraintes.

Nous avons aussi montré les avantages opérationnels qui peuvent être tirés de mécanismes de décision autonome à bord concernant les vidages : augmentation du nombre d'acquisitions vidées, diminution de l'âge de l'information sur les données vidées, utilisation possible de moins de stations de réception ou de moins de fenêtres de visibilité et augmentation possible de la taille des données acquises (fauchée plus large et/ou plus forte résolution de l'instrument d'observation).

Références

[1] E. Aarts and J. Lenstra, editors. *Local Search in Combinatorial Optimization*. John Wiley & Sons, 1997.

[2] R. Dechter, I. Meiri, and J. Pearl. Temporal Constraint Networks. *Artificial Intelligence*, 49 :61–95, 1991.

[3] F. Glover and M. Laguna. Tabu Search. In *Modern Heuristic Techniques for Combinatorial Problems*, pages 70–141. Blackwell Scientific Publishing, 1993.

[4] P. Van Hentenryck and L. Michel. *Constraint-based Local Search*. MIT Press, 2005.

[5] D. Joslin and D. Clements. Squeaky Wheel Optimization. *Journal of Artificial Intelligence Research*, 10 :353–373, 1999.

[6] H. Kellerer, U. Pferschy, and D. Pisinger. *Knapsack Problems*. Springer, 2004.

[7] S. Kirkpatrick, C. Gelatt, and M. Vecchi. Optimization by Simulated Annealing. *Science*, 220(4598) :671–680, 1983.

[8] M. Pinedo. *Scheduling : Theory, Algorithms, and Systems*. Springer, 2012.

[9] N. Policella, S. Smith, A. Cesta, and A. Oddi. Generating Robust Schedules through Temporal Flexibility. In *Proc. of the 14th International Conference on Automated Planning and Scheduling (ICAPS-04)*, pages 209–218, Whistler, Canada, 2004.

[10] C. Pralet and G. Verfaillie. Dynamic Online Planning and Scheduling using a Static Invariant-based Evaluation Model. In *Proc. of the 23rd International Conference on Automated Planning and Scheduling (ICAPS-13)*, Rome, Italy, 2013.

[11] C. Pralet and G. Verfaillie. Time-dependent Simple Temporal Networks : Properties and Algorithms. *RAIRO Operations Research*, 47(2) :173–198, 2013.

Nouvelles Clauses Bi-Assertives et leurs Intégration dans les Solveurs SAT Modernes

Said Jabbour[1] **Jerry Lonlac**[1,2] **Lakhdar Saïs**[1]

[1] CRIL - CNRS, Université d'Artois, Lens, France F-62307 Cedex
[2] Département d'Informatique - Université de Yaoundé 1 B.P. 812 Yaoundé, Cameroun
{jabbour,lonlac,sais}@cril.fr

Résumé

Dans ce papier, une nouvelle approche d'apprentissage de clauses est proposée. En effet, en traversant le graphe d'implications séparément à partir des littéraux x et $\neg x$, nous dérivons une nouvelle classe de clauses Bi-Assertives qui peuvent conduire à un graphe d'implications plus compact. Ces nouvelles clauses Bi-Assertives sont plus courtes et tendent à induire plus d'implications que les clauses Bi-Assertives classiques. Les résultats expérimentaux montrent que l'exploitation de cette nouvelle classe de clauses Bi-Assertives permet d'améliorer les performances des solveurs issus de l'état de l'art de SAT particulièrement sur les instances crafteds.

1 Introduction

Le problème SAT, i.e., le problème de décider si une formule booléenne sous forme normale conjonctive (CNF) est satisfiable ou non est central en Informatique et en Intelligence Artificielle incluant les problèmes de satisfaction de contraintes (CSP), planification, raisonnement non-monotone, vérification de logiciels, etc. Aujourd'hui, le problème SAT a gagné une audience considérable avec l'avénement d'une nouvelle génération de solveurs capablent de résoudre des grandes instances issues du codage des applications du monde réel ainsi que par le fait que ces solveurs constituent d'importants composants de base pour plusieurs domaines, e.g., SMT (SAT Modulo Théorie), démonstration automatique, comptage de modèles, problème QBF, etc. Ces solveurs communément appelés solveurs SAT modernes CDCL (Conflict Driven, Clause Learning) [9, 4] sont basées sur la propagation unitaire classique [2] finement combinée avec des structures de données efficaces (ex. watched literals), des politiques de redémarrages [6, 5, 7], des heuristiques de choix de variables basées sur les activités VSIDS (Variable State Independant, Decading Sum) [9], ainsi que l'apprentissage de clauses [8, 9, 13]. L'apprentissage de clauses est actuellement reconnu comme la composante la plus importante des solveurs SAT modernes. Ces solveurs SAT modernes peuvent être vus comme une version étendue de la bien connue procédure DPLL [2] obtenue grâce à ces différentes améliorations. L'algorithme pour la dérivation des clauses à partir des conflits dans ces solveurs SAT est basé sur l'utilisation d'une structure de données importante appelée graphe d'implications [8, 9]. Il est important de noter que la bien connue règle de résolution joue encore un rôle important dans l'efficacité des solveurs SAT modernes. Théoriquement, en intégrant l'apprentissage de clauses aux procédures DPLL [2], le solveur SAT obtenu formulé comme un système de preuve s'avère aussi puissant que la résolution générale [11, 12].

Une nouvelle classe de clauses assertives appelée clauses Bi-Assertives qui est une relaxation des clauses assertives a été proposée dans [10]. Ces clauses Bi-Assertives communément appelées clauses Bi-Assertives classiques permettent de découvrir les implications manquées par les clauses assertives classiques. Elle est définie de la même manière comme une clause assertive. Une clause asssertive contient exactement un littéral du niveau de conflit tandis qu'une clause Bi-Assertive contient exactement deux littéraux du niveau de conflit. Pour plus d'amples détails autours des clauses Bi-Assertives classiques, nous référons les lecteurs à [10].

L'approche proposée dans ce papier est basée sur une analyse de conflit séparée. En effet, en traversant le graphe d'implications séparément à partir de x et $\neg x$, nous dérivons une nouvelle classe de clauses Bi-Assertives. Ces clauses diffèrent de celles proposées par

Pipatsrisawat et Darwiche [10] car elles ne sont pas des clauses falsifiées c'est à dire satisfaites par l'interprétation partielle courante. Ces clauses ne peuvent être dérivées par le parcours traditionnel du graphe d'implications. De plus, nos clauses Bi-Assertives peuvent être utilisées pour obtenir un graphe d'implications plus compact. Elles sont plus courtes et tendent à induire plus d'implications que les clauses Bi-Assertives classiques.

Ce papier est organisé comme suit. Après quelques notations et définitions préliminaires, quelques notions théoriques autours des solveurs SAT, nous présentons notre approche. Finalement avant de conclure, les résultats expérimentaux obtenus sur les instances des récentes compétitions SAT démontrant la faisabilité de cette approche sont présentés.

2 Définitions

2.1 Définitions et notations préliminaires

Une *formule CNF* $\mathcal{F}$ est une conjonction de *clauses*, où une clause est une disjonction de *littéraux*. Un littéral est interprété comme une variable propositionnelle positive (x) ou négative ($\neg x$). Les deux littéraux x et $\neg x$ sont appélés *complémentaire*. On note par $\bar{l}$ le littéral complémentaire de l. Pour un ensemble de littéraux L, $\bar{L}$ est défini comme $\{\bar{l} \mid l \in L\}$. Une clause *unitaire* est une clause contenant seulement un seul littéral (appelé *littéral unitaire*), tandis qu'une clause binaire contient excatement deux littéraux. Une *clause vide* , notée $\bot$, est interprétée comme fausse (insatisfiable), alors qu'une *formule CNF vide*, notée $\top$, est interprétée comme vraie (satisfiable). L'ensemble des variables apparaissant dans $\mathcal{F}$ est noté $V_{\mathcal{F}}$. Un ensemble de littéraux est *complet* s'il contient un littéral pour chaque variable de $V_{\mathcal{F}}$, et *fondamental* s'il ne contient pas de littéraux complémentaires. Une *affectation* ρ d'une formule booléenne $\mathcal{F}$ est une fonction qui associe la valeur $\rho(x) \in \{false, true\}$ à quelques variables de $x \in \mathcal{F}$. ρ est dite *complète* s'elle attribue une valeur pour chaque variable $x \in \mathcal{F}$, et *partielle* sinon. Une affectation est alternativement représentée par un ensemble de littéraux complet et fondamental, de façon évidente un *modèle* d'une formule $\mathcal{F}$ est une affectation ρ qui laisse la formule *vraie* ; notée $\rho \models \Sigma$.

On dénote par $\eta[x, c_i, c_j]$ la *résolvante* entre une clause c_i contenant le littéral x et c_j une clause contenant l'opposé de ce même littéral $\neg x$. En d'autres termes, $\eta[x, c_i, c_j] = c_i \cup c_j \backslash \{x, \neg x\}$.

Nous dénotons également par $\mathcal{F}|_x$ la formule obtenue en assignant à vrai le littéral x dans $\mathcal{F}$. Formellement, $\mathcal{F}|_x = \{c \mid c \in \mathcal{F}, \{x, \neg x\} \cap c = \emptyset\} \cup \{c \backslash \{\neg x\} \mid c \in \mathcal{F}, \neg x \in c\}$ (c'est à dire : les clauses contenant x sont par conséquent satisfaites et enlevées ; et celles contenant $\neg x$ sont simplifiées). Nous définissons $\mathcal{F}^*$ comme la formule $\mathcal{F}$ fermée sous la propagation unitaire, définie de manière récursive comme suit : (1) $\mathcal{F}^* = \mathcal{F}$ si $\mathcal{F}$ ne contient pas de clause unitaire,, (2) $\mathcal{F}^* = \bot$ si $\mathcal{F}$ contient deux clauses unitaires $\{x\}$ et $\{\neg x\}$, (3) sinon, $\mathcal{F}^* = (\mathcal{F}|_x)^*$ où x est le littéral apparaissant dans une clause unitaire de $\mathcal{F}$. Une clause c est déduite par propagation unitaire de $\mathcal{F}$, notée $\mathcal{F} \vdash c$, si $(\mathcal{F}|_{\bar{c}})^* = \bot$.

2.2 Recherche DPLL

DPLL [2] est une procédure de recherche de type *backtrack* ; À chaque nœud les littéraux affectés (le littéral de décision et les littéraux propagés) sont étiquetés avec le même *niveau de décision*, initialisé à 1 et incrémenté à chaque nouveau point de décision. Le niveau de décision courant est le niveau le plus élevé dans la pile de propagation. Lors d'un retour-arrière ("*backtrack*"), les variables ayant un niveau supérieur au niveau du backtrack sont défaites et le niveau de décision courant est décrémenté en conséquence (égal au niveau du backtrack). Au niveau i, l'interprétation partielle courante ρ peut être représentée comme une séquence décision-propagations de la forme $\langle (x_k^i), x_{k_1}^i, x_{k_2}^i, \ldots, x_{k_{n_k}}^i \rangle$ où le premier littéral x_k^i correspond au littéral de décision x_k affecté au niveau i et chaque $x_{k_j}^i$ de l'ensemble $1 \leq j \leq n_k$ représente les littéraux unitaires propagés à ce même niveau i. Soit $x \in \rho$, on note $l(x)$ le niveau d'affectation de x. Pour une clause α, $l(\alpha)$ est défini comme le maximum des niveaux de ses littéraux affectés.

2.3 Analyse de conflit et graphe d'implications

Le graphe d'implications est une représentation standard utilisée classiquement pour analyser les conflits dans les solveurs SAT modernes. À chaque fois qu'un littéral y est propagé, on sauvegarde une référence de la clause impliquant la propagation de y, qu'on note $imp(y)$. La clause $imp(y)$, appelée implication de y, est dans ce cas de la forme $(x_1 \vee \cdots \vee x_n \vee y)$ où chaque littéral x_i est faux sous l'affectation partielle courante ($\rho(x_i) = false, \forall i \in 1..n$), alors que $\rho(y) = vraie$. Lorsqu'un littéral y n'est pas obtenu par propagation mais issue d'une décision, $imp(y)$ est indéfinie, qu'on note par convention $imp(y) = \bot$. Quand $imp(y) \neq \bot$, on dénote par $exp(y)$ l'ensemble $\{\overline{x} \mid x \in imp(y) \setminus \{y\}\}$, appelé ensemble des *explications* de y. Quand $imp(y)$ est indéfini, on défini $exp(y)$ comme l'ensemble vide. Dans la suite, nous rappelons la définition formelle d'un graphe d'implications, des

clauses assertives classiques et comment ces clauses sont dérivées [1].

Definition 1 (graphe d'implications) *Soient $\mathcal{F}$ une formule* CNF*, ρ une interprétation partielle. Soit exp l'ensemble des explications pour des littéraux déduits par propagation unitaire dans ρ. Le graphe d'implications associé à $\mathcal{F}$, ρ et exp est $\mathcal{G}_{\mathcal{F}}^{\rho,exp} = (\mathcal{N}, \mathcal{E})$ où :*

— *$\mathcal{N} = \rho$, c'est-à-dire un nœud pour chaque littéral de ρ, de décision ou propagé ;*

— *$\mathcal{E} = \{(x, y) \mid x \in \rho, y \in \rho, x \in \exp(y)\}$.*

Dans le reste du papier, pour des raisons de simplicité, un graphe d'implications sera simplement noté $\mathcal{G}_{\mathcal{F}}^{\rho}$. on note par m le niveau du conflit.

Exemple 1 *$\mathcal{G}_{\mathcal{F}}^{\rho}$, représenté par la figure 1 est un graphe d'implications pour la formule $\mathcal{F}$ et l'interprétation partielle ρ données ci-dessous : $\mathcal{F} \supseteq \{c_1, \ldots, c_{12}\}$*

$(c_1) \neg x_1 \vee \neg x_{11} \vee x_2$ $\quad$ $(c_2) \neg x_1 \vee x_3$
$(c_3) \neg x_2 \vee \neg x_{12} \vee x_4$ $\quad$ $(c_4) \neg x_1 \vee \vee \neg x_3 \vee x_5$
$(c_5) \neg x_4 \vee \neg x_5 \vee \neg x_6 \vee x_7$ $\quad$ $(c_6) \neg x_5 \vee \neg x_6 \vee x_8$
$(c_7) \neg x_7 \vee x_9$ $\quad$ $(c_8) \neg x_5 \vee \neg x_8 \vee \neg x_9$
$(c_9) \neg x_{10} \vee \neg x_{17} \vee x_1$ $\quad$ $(c_{10}) \neg x_{13} \vee \neg x_{14} \vee x_{10}$
$(c_{11}) \neg x_{13} \vee x_{17}$ $\quad$ $(c_{12}) \neg x_{15} \vee \neg x_{16} \vee x_{13}$

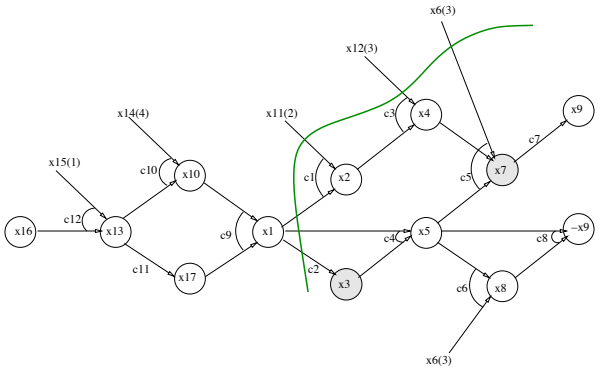


FIGURE 1 – Graphe d'implications $\mathcal{G}_{\mathcal{F}}^{\rho} = (\mathcal{N}, \mathcal{E})$

$\rho = \{\langle \ldots x_{15}^1 \ldots \rangle \langle (x_{11}^2) \ldots\ldots \rangle \langle (x_{12}^3) \ldots x_6^3 \ldots \rangle$
$\langle (x_{14}^4), \ldots \rangle \langle (x_{16}^5), x_{13}^5, x_{10}^5, x_{17}^5, x_1^5 \ldots \rangle\}$. *le niveau de conflit est 5 et $\rho(\mathcal{F}) = faux$.*

Definition 2 (clause assertive) *Une clause c de la forme $(\alpha \vee x)$ est appelée clause assertive si et seulement si $\rho(c) =$ false, $l(x) = m$ et $\forall y \in \alpha, l(y) < l(x)$. x est appelé littéral* assertif*, tandis que $assertingLevel(c) = max\{l(\neg y) \mid y \in \alpha\}$ est le niveau d'assertion du littéral x.*

L'analyse de conflit est le résultat de l'application de la résolution à partir de la clause conflit en utilisant différentes implications encodées dans le graphe d'implications. On appelle cela, la dérivation de la clause assertive (DCA en court).

Definition 3 (dérivation de la clause assertive) *Une dérivation de la clause assertive π est une sequence de clauses $\langle \sigma_1, \sigma_2, \ldots \sigma_k \rangle$ satisfaisant les conditions suivantes :*

1. *$\sigma_1 = \eta[x, imp(x), imp(\neg x)]$, où $\{x, \neg x\}$ est un conflit ;*
2. *σ_i, pour tout $i \in 2..k$, est construit en sélectionnant un littéral $y \in \sigma_{i-1}$ pour lequel $imp(\overline{y})$ est défini. Nous avons alors $y \in \sigma_{i-1}$ et $\overline{y} \in imp(\overline{y})$, les deux clauses résolues. La clause σ_i est définie comme $\eta[y, \sigma_{i-1}, imp(\overline{y})]$;*
3. *σ_k est en plus une clause assertive.*

Considérons à nouveau l'exemple 1. Le parcours du graphe d'implications $\mathcal{G}_{\mathcal{F}}^{\rho}$ (voir Figure 1) conduit à la dérivation de la clause assertive $\langle \sigma_1, \ldots, \sigma_7 \rangle$ où :

— $\sigma_1 = \eta[x_9, c_7, c_8] = (\neg {x_5}^5 \vee \neg {x_7}^5 \vee \neg {x_8}^5)$
— $\sigma_2 = \eta[x_8, \sigma_1, c_6] = (\neg x_6^3 \vee \neg {x_5}^5 \vee \neg {x_7}^5)$
— $\sigma_3 = \eta[x_7, \sigma_2, c_5] = (\neg x_6^3 \vee \neg {x_5}^5 \vee \neg {x_4}^5)$
— $\sigma_4 = \eta[x_4, \sigma_3, c_3] = (\neg x_{12}^3 \vee \neg x_6^3 \vee \neg {x_5}^5 \vee \neg {x_2}^5)$
— $\sigma_5 = \eta[x_5, \sigma_4, c_4] = (\neg x_{12}^3 \vee \neg x_6^3 \vee \neg {x_2}^5 \vee \neg {x_3}^5 \vee \neg x_1^5)$
— $\sigma_6 = \eta[x_2, \sigma_5, c_1] = (\neg x_{11}^2 \vee \neg x_{12}^3 \vee \neg x_6^3 \vee \neg {x_3}^5 \vee \neg x_1^5)$
— $\sigma_7 = \eta[x_3, \sigma_6, c_2] = (\neg {x_{11}}^2 \vee \neg {x_{12}}^3 \vee \neg {x_6}^3 \vee \neg {x_1}^5)$

La clause σ_7 est la prémière clause rencontrée qui contient un seul littéral $\neg x_1$ du niveau de conflit courant 5. Notons que cette résolvante est fausse sous l'interprétation partielle ρ. Par conséquent, le littéral $\neg x_1$ est impliqué au niveau 3 qui correspond au niveau maximum de tous les autres littéraux ($\neg {x_{11}}^2, \neg {x_{12}}^3$ et $\neg {x_6}^3$) de σ_7. Les solveurs SAT ajoutent la clause σ_7 à la base des clauses apprises, effectuent un retour-arrière non chronologique au niveau 3 et affectent le littéral assertif $\neg x_1$ à la valeur de vérité vraie par propagation unitaire. Le nœud x_1 correspondant au littéral assertif $\neg x_1$ est appelé Premier Unique Point d'Implication (First UIP).

Nous introduisons à présent une propriété importante appelée *1−empowerment* [10], qui caractérise les clauses assertives classiques.

Definition 4 (1−Empowerment) *Soit c une clause de la forme $(\alpha \vee l)$, où l est un littéral et α une disjonction de littéraux (une sous-clause). La clause c est* 1−empowerment *par rapport à la formule* CNF *$\mathcal{F}$ via le littéral l si et seulement si :*

1. $\mathcal{F} \models (\alpha \vee l)$ *: La clause c est impliquée par* $\mathcal{F}$
2. $\mathcal{F} \wedge \neg\alpha \nvdash l$ *: Le littéral l ne peut pas être dérivé de la formule* $\mathcal{F} \wedge \neg\alpha$ *en utilisant la propagation unitaire.*

Toute clause assertive $c = (\alpha \vee l)$ satisfait la propriété *1−empowerment*. En effet, comme c est dérivée de la formule $\mathcal{F}$ par résolution, alors c est une conséquence de $\mathcal{F}$ (condition 1 de la définition 4). La seconde condition de la définition 4 est aussi satisfaite comme le littéral l n'est pas dérivé au niveau d'assertion où la sous-clause α est falsifiée (condition 2 de la définition 4). Cette propriété exprime que le littéral l ne peut pas être déduit par propagation unitaire de $\mathcal{F}$ sans l'ajout de la clause c à $\mathcal{F}$. Cependant, si on ajoute la clause assertive à la base des clauses apprises, on déduit par propagation unitaire que le littéral l doit être assigné à vrai au niveau d'assertion. Pour plus de détails sur les shémas d'apprentissage basés sur l'architecture CDCL, nous référons les lecteurs à [8, 9, 1].

Un autre type de clauses assertives appelées clauses Bi-Assertives fut introduit dans [10].

Definition 5 (Clauses Bi-Assertives) *Une clause c de la forme* $(\alpha \vee x \vee y)$ *est appelée clause Bi-Assertive si* $\rho(c) = false$, $l(\alpha) < m$ *et* $level(y) = level(x) = m$.

Il est important de noter qu'une clause Bi-Assertive classique est fausse sous l'interprétation courante ρ. Pour dériver de telles clauses Bi-Assertives à partir d'un conflit, on suit exactement la même dérivation par résolution. La seule différence est que le processus se termine quand la résolvante courante est une clause Bi-Assertive. Comme toute clause Bi-Assertive ne contribue pas à la progapation unitaire, dans [10], les auteurs proposent d'apprendre la première clause Bi-Assertive qui satisfait la propriété *1−empowerment* respectivement avec les clauses utilisées pour sa dérivation.

Exemple 2 *Considérons par exemple la formule* CNF $\mathcal{F} = (x_1 \vee x_2) \wedge (\neg x_1 \vee x_3) \wedge (x_2 \vee \neg x_3 \vee x_4)$ *et la clause* $(x_2 \vee x_4)$ *qui est impliquée par la formule* $\mathcal{F}$. *L'ajout du littéral* $\neg x_4$ *à* $\mathcal{F}$ *ne permet pas à la propagation unitaire de dériver* x_2 *bien que* x_2 *soit impliqué par* $\mathcal{F} \wedge \neg x_4$. *Cette dérivation devient possible si nous ajoutons la clause* $(x_2 \vee x_4)$ *à* $\mathcal{F}$. *Ainsi la clause* $(x_2 \vee x_4)$ *est* 1−empowering *respectivement avec* $\mathcal{F}$ *via le littéral* x_2. *Cependant la clause* $(x_2 \vee x_3)$ *n'est pas* 1−empowering *respectivement avec la formule* $\mathcal{F}$, *car* $\mathcal{F} \wedge \neg x_2 \vdash x_3$ *et* $\mathcal{F} \wedge \neg x_3 \vdash x_2$.

Dans la section suivante, nous proposons une nouvelle classe de clauses Bi-Assertives plus courtes et qui tendent à induire plus d'implications que les clauses Bi-Assertives classiques. Ces nouvelles clauses Bi-Assertives conduisent à un graphe d'implications plus compact.

3 Une nouvelle classe de clauses Bi-Assertives

Dans cette section, nous montrons premièrement comment dériver les nouvelles clauses Bi-Assertives. Ensuite, nous démontrons que leur utilisation peut conduire à obtenir plus d'implications qu'avec les clauses Bi-Assertives classiques. Illustrons cette nouvelle approche proposée en utilisant un simple exemple.

3.1 Motivation

Soit $\mathcal{F}$ une formule CNF, ρ une affectation partielle. Supposons que nous avons un graphe d'implications $\mathcal{G}^{\rho}_{\mathcal{F}}$ associé à $\mathcal{F}$ et ρ. Pour des raisons de simplicité, la figure 2 représente le graphe d'implications restreint au sous-graphe induit par les nœuds entre le conflit et le Premier Unique Point d'Implication (First UIP). Supposons que le niveau de conflit est 5.

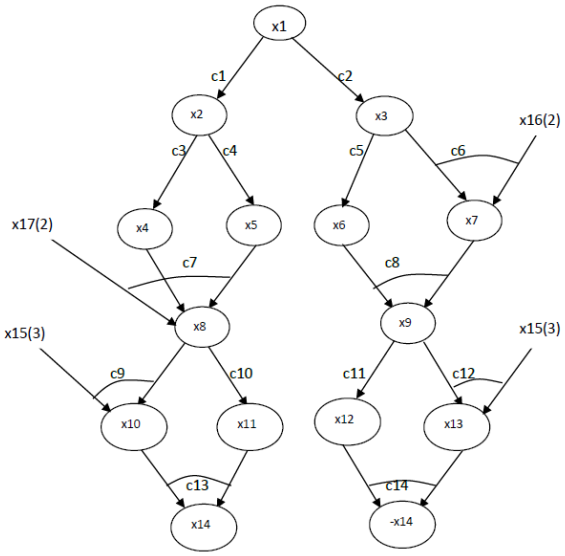


FIGURE 2 – Sous-graphe d'implications $\mathcal{G}^{\rho}_{\mathcal{F}} = (\mathcal{N}, \mathcal{E})$

En traversant le graphe d'implications séparément à partir des littéraux x_{14} et $\neg x_{14}$ jusqu'au Premier Unique Point d'Implication x_1, nous dérivons de nouvelles clauses Bi-Assertives δ_1, δ_2, δ_3 et δ_4 comme suit :

— $\sigma_1 = \eta[x_{10}, c_{13}, c_9] = \neg x_{15} \vee \neg x_{11} \vee \neg x_8 \vee x_{14}$
— $\delta_1 = \eta[x_{11}, \sigma_1, c_{10}] = \neg x_{15} \vee \neg x_8 \vee x_{14}$
— $\sigma_2 = \eta[x_4, c_7, c_3] = \neg x_{17} \vee \neg x_5 \vee \neg x_2 \vee x_8$
— $\delta_2 = \eta[x_5, \sigma_2, c_4] = \neg x_{17} \vee \neg x_2 \vee x_8$

— $\sigma_3 = \eta[x_{12}, c_{14}, c_{11}] = \neg x_{13} \vee \neg x_9 \vee \neg x_{14}$
— $\delta_3 = \eta[x_{13}, \sigma_3, c_{12}] = \neg x_{15} \vee \neg x_9 \vee \neg x_{14}$
— $\sigma_4 = \eta[x_6, c_8, c_5] = \neg x_3 \vee \neg x_7 \vee x_9$
— $\delta_4 = \eta[x_7, \sigma_4, c_6] = \neg x_{16} \vee \neg x_3 \vee x_9$

Les nouvelles clauses Bi-Assertives δ_1, δ_2, δ_3, δ_4 forment un ensemble de clauses connexes. Plus précisement, la clause δ_2 est liée à la clause δ_1 par la variable x_8, δ_1 est liée à δ_3 par la variable x_{14} et δ_3 est liée à δ_4 par la variable x_9. Comme nous pouvons l'observer, l'ensemble des clauses Bi-Assertives forment une chaîne de clauses connectées. Il est important de noter que les clauses Bi-Assertives classiques qui peuvent être obtenues du graphe d'implications de la figure 2 sont :

— $\delta'_1 = \neg x_{15} \vee \neg x_8 \vee \neg x_9$,
— $\delta'_2 = \neg x_{15} \vee \neg x_{17} \vee \neg x_2 \vee \neg x_9$,
— $\delta'_3 = \neg x_{15} \vee \neg x_{16} \vee \neg x_8 \vee \neg x_3$,
— $\delta'_4 = \neg x_{15} \vee \neg x_{17} \vee \neg x_{16} \vee \neg x_2 \vee \neg x_3$.

Notons qu'en utilisant le schéma d'apprentissage classique de clauses (schéma First UIP), on peut dériver la clause assertive $(\neg x^2_{17} \vee \neg x^2_{16} \vee \neg x^3_{15} \vee \neg x^5_1)$ qui fournit un niveau de backjumping égal à 3.

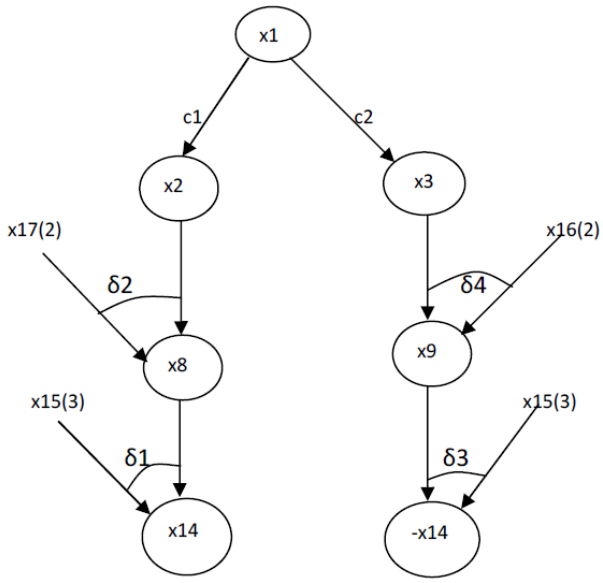


FIGURE 3 – Nouveau graphe d'implications plus compact obtenu de $\mathcal{G}^{\rho}_{\mathcal{F}} = (\mathcal{N}, \mathcal{E})$ en utilisant les nouvelles clauses Bi-Assertives.

A travers la figure 3, nous montrons que le graphe d'implications original (figure 2) peut être reécrit de façon plus compact.

Considérons à présent que les nouvelles clauses Bi-Assertives δ_1, δ_2, δ_3 et δ_4 sont ajoutées à la base des clauses apprises et supposons qu'au niveau 3 le littéral x_2 est assigné à vrai. Il est facile de voir que tous les littéraux x_8, x_{14}, $\neg x_9$, $\neg x_3$ et $\neg x_1$ (voir figure 3) ensemble avec les littéraux x_4, x_5, x_{10}, x_{11} seront impliqués par propagation unitaire. Ces derniers littéraux propagés sont impliqués grâce aux clauses initiales de la formule qui sont c_3, c_4, c_9 et c_{10}.
Cependant, si nous considérons le graphe d'implications original (figure 2), nous dérivons par propagation unitaire uniquement les littéraux x_8, x_{14}, x_4, x_5, x_{10}, x_{11} et pas les littéraux $\neg x_9$, $\neg x_3$ et $\neg x_1$.

Supposons maintenant que toutes les clauses Bi-Assertives classiques δ'_1, δ'_2, δ'_3 et δ'_4 sont ajoutées à la base des clauses apprises. L'affectation de x_2 à vrai au niveau 3 conduit au même ensemble d'implications par propagation unitaire. Cependant, si nous décidons d'assigner x_{14} à vrai au niveau 3, en utilisant les nouvelles clauses Bi-Assertives, nous dérivons les littéraux $\neg x_9$, $\neg x_3$ et $\neg x_1$ tandis qu'avec les clauses Bi-Assertives classiques aucun littéral n'est impliqué par propagation unitaire.

Autre différence importante qui peut être faite ici est que la dérivation de toutes les clauses Bi-Assertives classiques possibles est quadratique dans le pire des cas alors que les nouvelles clauses Bi-Assertives peuvent être dérivées en temps linéaire (voir figure 3). Par conséquent, la recherche de toutes les clauses Bi-Assertives classiques prend plus de temps que la recherche des nouvelles clauses Bi-Assertives proposées.

A partir de cet exemple illustratif, on voit que l'ajout des nouvelles clauses Bi-Assertives à la base des clauses apprises permet de dériver plus d'implications que l'ajout des clauses Bi-Assertives classiques. Nous pouvons également observer que les clauses Bi-Assertives proposées établissent un lien entre le littéral assertif $\neg x_1$ et les deux littéraux conflits x_{14} et $\neg x_{14}$.

3.2 Analyse de conflit séparée : formulation générale

On donne à présent une présentation formelle de notre approche de dérivation des nouvelles clauses Bi-Assertives. Dans toutes les définitions suivantes, nous considérons m comme le niveau de conflit courant.

Dans notre approche, à partir d'un simple conflit, nous dérivons plusieurs nouvelles clauses Bi-Assertives en traversant le graphe d'implications séparement à partir des deux littéraux conflits x et $\neg x$ jusqu'au Premier Unique Point d'Implications (First UIP). Ces nouvelles clauses sont ensuite ajoutées à la base des clauses apprises pour intensifier la propagation unitaire.

Notons que nous avons seulement besoin d'un simple parcours du graphe d'implications pour dériver à la fois la clause assertive classique et l'ensemble des nouvelles clauses Bi-Assertives. En effet, la clause assertive classique peut être progressivement générée des

nouvelles clauses Bi-Assertives. De cette manière, nous n'avons aucun surcoût de recherche additionnel dû à la dérivation de ces nouvelles clauses Bi-Assertives.

Definition 6 (Dérivation de la clause Bi-Assertive) *Soit x un littéral conflit, nous définissons la dérivation de la clause Bi-Assertive à partir de x comme la séquence de clauses $C_x = \langle \sigma_1^x, \sigma_2^x, \ldots, \sigma_k^x \rangle$ satisfaisant les conditions suivantes :*

1. *σ_1^x, est dérivée par résolution sur z entre $imp(x)$ et $imp(z)$ où $\neg z \in imp(x)$;*
2. *σ_i^x, pour tout $i \in 2..k$, est construite par résolution sur un littéral $z \in \sigma_{i-1}^x$ s.t. $l(z) = m$ et pour lequel $imp(\overline{z})$ est définit. La clause σ_i^x est définie comme $\eta[z, \sigma_{i-1}^x, imp(\overline{z})]$;*
3. *σ_k^x est une clause Bi-Assertive.*

Definition 7 (littéral Bi-Assertif) *Soit x un littéral conflit et $C_x = \langle \sigma_1^x, \sigma_2^x, \ldots, \sigma_k^x \rangle$ une dérivation de la clause Bi-Assertive. Nous définissons un littéral Bi-Assertif comme un littéral $b \in \sigma_k^x$ tel que $b \neq x$ et $l(b) = m$.*

Si nous considérons à nouveau le graphe d'implications de la figure 2, l'ensemble des différents littéraux Bi-Assertifs sera $\{x_8, x_9, x_2, x_3, x_1\}$.

Propriété 1 *Soient $\mathcal{G}$ un graphe d'implications et x un littéral conflit de $\mathcal{G}$ Soient B^x et $B^{\neg x}$ les deux ensembles de clauses Bi-Assertives générées à partir des deux littéraux conflits x et $\neg x$ respectivement. Au niveau du backjumping, si un littéral Bi-Assertif y de B^x (respectivement $B^{\neg x}$) est assigné à vrai, alors tous les littéraux $\neg z$ tels que z est un littéral Bi-Assertif de $B^{\neg x}$ (respectivement B^x) seront impliqués.*

Preuve : La preuve est triviale. Une illustration de la preuve est donnée dans l'exemple de motivation précédent.

Il est facile de voir que le nombre de nouvelles clauses Bi-Assertives qui peuvent être générées de tout graphe d'implications est au moins égal à 2. En effet, si nous considérons un graphe d'implications $\mathcal{G}$ avec un littéral conflit x et y comme littéral assertif (schéma First UIP), il est facile de voir qu'on peut dériver au moins deux clauses Bi-Assertives $b_1 = (\alpha \vee y \vee x)$ et $b_2 = (\beta \vee y \vee \neg x)$. Dans ce cas la clause assertive résultante est $a = (\alpha \vee \beta \vee y)$.

Propriété 2 *Soit $\mathcal{G}$ un graphe d'implications, x un littéral conflit de $\mathcal{G}$. Soient N^x et $N^{\neg x}$ les deux ensembles de clauses encodées dans $\mathcal{G}$ entre le First UIP et les deux littéraux conflits x et $\neg x$ respectivement. La complexité temporelle nécessaire pour générer toutes nos nouvelles clauses Bi-Assertives est en $\mathcal{O}(|N^x| + |N^{\neg x}|)$ dans le pire des cas.*

Preuve : Comme nos nouvelles clauses Bi-Assertives sont générées en remontant en parallèle les deux branches (branche contenant le littéral x et celle contenant $\neg x$) du graphe d'implications $\mathcal{G}$ à partir des deux littéraux conflits x et $\neg x$, chaque clause de N^x et $N^{\neg x}$ est impliquée dans une étape de résolution. Ainsi la complexité temporelle nécessaire pour la dérivation de toutes les nouvelles clauses Bi-Assertives est en $\mathcal{O}(|N^x| + |N^{\neg x}|)$ dans le pire des cas.

Si nous considérons les clauses Bi-Assertives classiques [10], et souhaitons générer toutes ces clauses, la complexité temporelle nécessaire pour effectuer cette opération sera quadratique dans le pire des cas.

Propriété 3 *Soit $\mathcal{G}$ un graphe d'implications, x un littéral conflit de $\mathcal{G}$. Soient N^x et $N^{\neg x}$ les deux ensembles de clauses encodées dans $\mathcal{G}$ entre le First UIP et les deux littéraux conflits x et $\neg x$ respectivement. La complexité temporelle nécessaire pour générer toutes les clauses Bi-Assertives classiques est en $\mathcal{O}(|N^x| \times |N^{\neg x}|)$ dans le pire des cas.*

Preuve : Le pire des cas est rencontré lorsque que nous avons un graphe d'implications $\mathcal{G}$ avec $|N^x|$ et $|N^{\neg x}|$ littéraux Bi-Assertifs. La forme d'un tel graphe est représentée par la figure 3. Cette figure correspond à un graphe d'implications dans lequel tous les nœuds sont des littéraux Bi-Assertifs. Comme le processus de génération des clauses Bi-Assertives classiques commence par effectuer une prémière étape de résolution entre les deux clauses impliquant les littéraux x et $\neg x$, pour générer toutes les clauses Bi-Assertives classiques, nous gardons alternativement le littéral Bi-Assertif de la partie du graphe liée à x (respectivement à $\neg x$), et nous traversons toute l'autre partie du graphe liée à $\neg x$ (respectivement à x). Par conséquent, la complexité temporelle nécessaire pour la dérivation de toutes les clauses Bi-Assertives **classiques** est en $\mathcal{O}(|N^x| \times |N^{\neg x}|)$.

Il est important de noter que les clauses Bi-Assertives classiques et les nouvelles clauses Bi-Assertives proposées dans ce papier diffèrent sur certains aspects. Premièrement, les nouvelles clauses Bi-Assertives (respectivement clauses Bi-Assertives classiques) sont satisfaites (respectivement falsifiées) sous l'interprétation courante. Deuxièmement, les deux classes de clauses sont générées en utilisant des traversées différentes du graphe d'implications. Par conséquent, les ensembles de clauses Bi-Assertives générées par les deux approches sont différents (Voir l'exemple de motivation donné à la section 3.1).

3.3 Exploitation des nouvelles clauses Bi-Assertives

Au cours de la recherche, quand un conflit est rencontré sur un littéral x, nous l'analysons en traversant le graphe d'implications séparément à partir de x et $\neg x$. Toutes les clauses Bi-Assertives dérivées au cours de cette analyse sont stockées dans une nouvelle base de clauses Bi-Assertives apprises. Ces nouvelles clauses sont gérées de la même manière que les clauses de la base des clauses apprises classiques. Nous exploitons une stratégie de réduction de la base des clauses apprises identique à celle implémentée dans `Minisat` 2.2 [4] qui est le solveur le plus utilisé actuellement. En effet, dans ce solveur, la fonction de réduction de la base des clauses apprises est appelée une fois que la taille de la base dépasse un certain seuil. La clause assertive classique est aussi ajoutée à la base des clauses apprises classiques.

4 Expérimentations

Les expérimentations ont été réalisées sur un large panel d'instances industrielles et crafteds issues des dernières compétitions SAT. Toutes les instances sont simplifiées par pré-traitement `SatElite` [3]. Afin d'étudier l'impact de notre approche, nous l'avons implémentée dans le solveur `Minisat` 2.2 [4] et nous avons effectué une comparaison entre le solveur original et celui amélioré avec l'apprentissage des nouvelles clauses Bi-Assertives que nous appelons `Minisat+NB`. Tous les tests ont été effectués sur un cluster Xeon 3.2GHz (32 GB RAM). Les résultats du temps de calcul sont indiqués en secondes. Pour ces expérimentations, le temps de calcul limite a été fixé à 4 heures.

4.1 Problèmes crafteds

Pour ces problèmes, nous utilisons un ensemble d'instances crafteds issues de la compétition SAT 2011. Le schéma (en échelle logarithmique) donné dans la partie haute de la figure 4 détaille les résultats de `Minisat` et `Minisat+NB` sur chaque instance crafted. L'axe x (respectivement y) correspond au temps CPU tx (respectivement ty) obtenu par `Minisat` (respectivement `Minisat+NB`).

Chaque point de coordonnées (tx, ty) correspond à une instance SAT. Les points en-dessous (respectivement au-dessus) la diagonale indique les instances résolues plus rapidement en utilisant notre approche c'est à dire $ty < tx$ (respectivement $ty > tx$). La majorité des points sur la figure se trouvent en-dessous la diagonale, ce qui signifie que `Minisat+NB` est meilleur. Cette figure montre clairement le gain de temps obtenu grâce à notre approche. Globalement, sur les instances de type crafted, notre approche apporte des améliorations intéressantes.

La partie bas de la figure 4 montre les mêmes résultats avec une représentation différente donnant pour chaque technique le nombre d'instances résolues (# instances) en moins de t secondes. Cette figure confirme l'efficacité de notre approche d'apprentissage sur ces problèmes. De cette figure, nous pouvons observer que `Minisat+NB` est généralement plus rapide et résoud 15 instances de plus que `Minisat`.

Une analyse plus fine de la partie haute de la figure 4 montre que `Minisat+NB` résoud plus rapidement 81 instances que `Minisat`, qui lui même résoud 51 instances plus efficacement que son opposé.

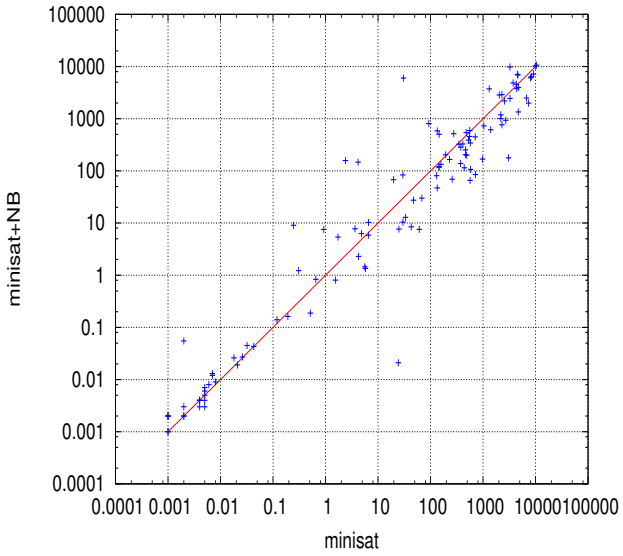
4.2 Problèmes industriels

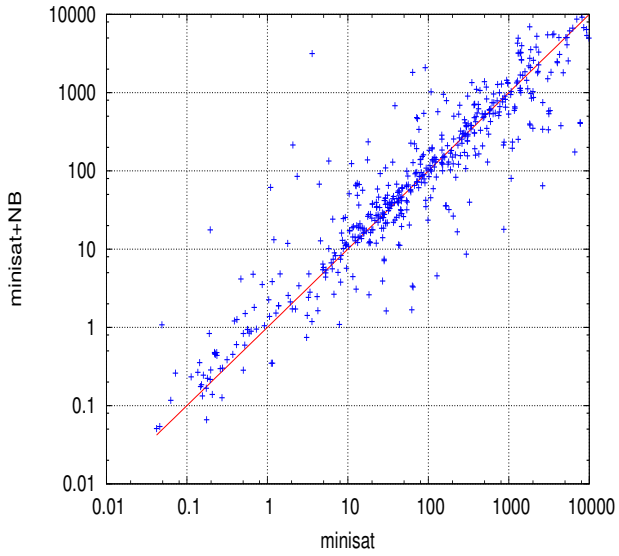
Pour ces problèmes, nous utilisons un ensemble d'instances industrielles issues de la SAT Challenge 2012. Le schéma (en échelle logarithmique) représenté par la figure 5 détaille les résultats de `Minisat` et `Minisat+NB` sur chaque instance industrielle. Sur ces instances industrielles, les résultats montrent que les deux solveurs ont un comportement similaire. Ceci confirme que lorsque le graphe d'implications ne contient pas de littéraux Bi-Assertifs intermédiaires sur les deux côtés des littéraux conflit, notre approche ne cause pas de surcoût additionnel. En effet, le graphe d'implications est traversé seulement une fois. Globalement, nous pouvons voir que l'addition de notre processus d'apprentissage à `Minisat` améliore ses performances sur certaines familles d'instances.

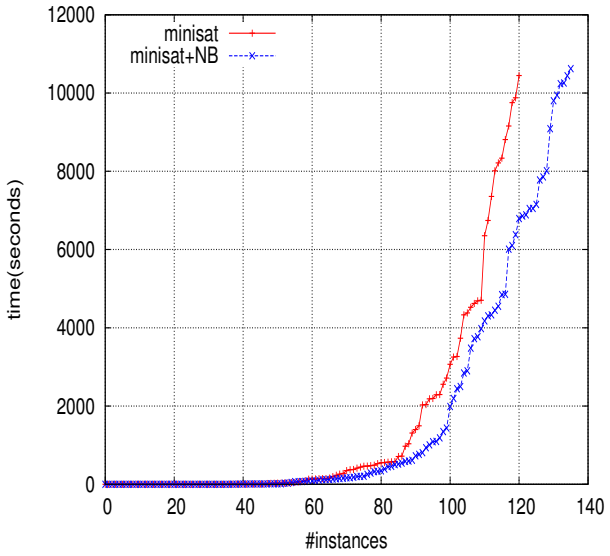
La table 1 représente un focus sur quelques familles industrielles. Pour ces familles, les gains sont relativement importants. Par exemple, si nous conséderons la famille *vmpc*, nous pouvons voir que notre approche d'apprentissage permet d'avoir un gain d'un ordre de magnitude avec `Minisat+NB` (instances 31, and 33). Sur la même famille, `Minisat+NB` résoud une instance de plus que `Minisat`. Sur la famille *manol*, nous pouvons également voir que `Minisat+NB` résoud 6 instances de plus que `Minisat`. Sur la famille *velev*, `Minisat+NB` est meilleur sur les 4 instances résolues par `Minisat+NB` et `Minisat` et résoud 3 instances de plus que `Minisat`. Globalement, on peut voir que sur certaines familles, `Minisat+NB` est plus rapide et résoud plus de problèmes que `Minisat`.

5 Remerciements

Ce travail a été soutenu par l'Agence Nationale de la Recherche Française sur le projet TUPLES- ANR programme blanc 10-BLAN-0210 "Tractability for Understanding and Pushing forward the Limits of Effi-







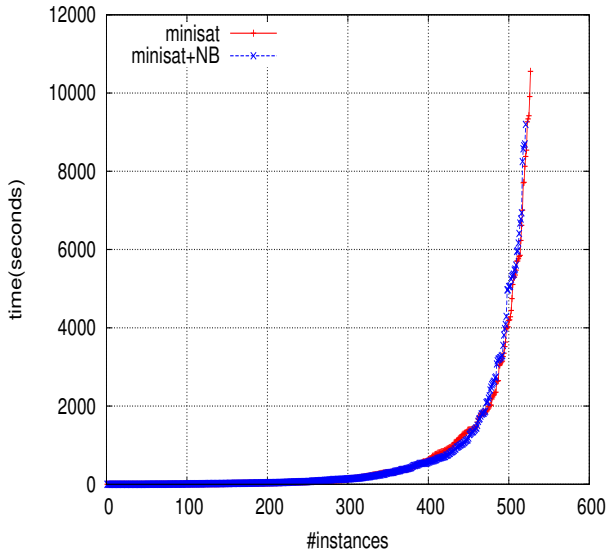


FIGURE 4 – **Problèmes Crafted : Minisat vs Minisat+NB**

FIGURE 5 – **Problèmes Industriels : Minisat vs Minisat+NB**

familles	Minisat	Minisat+NB
vmpc_26	**14.14**	21.06
vmpc_27	42.62	**11.31**
vmpc_28	98.99	**23.45**
vmpc_29	**73.41**	465.88
vmpc_30	**467.25**	535.62
vmpc_31	7707.81	**415.29**
vmpc_32	**1822.31**	2742.98
vmpc_33	6617.32	**174.37**
vmpc_34	-	**120.65**
manol-pipe-f7idw	4442.4	**378.08**
manol-f9b	**125.13**	139.83
manol-f9bidw	-	**670.52**
manol-f9nidw	-	**680.25**
manol-f8nidw	3105.14	**335.41**
manol-f8idw	-	**1606.05**
manol-f8bidw	-	**181.76**
manol-fn	**36.11**	44.09
manol-f10ni	**392.21**	492.48
manol-f10bi	491.96	**490.90**
manol-f10n	**291.57**	296.35
manol-f10nidw	-	**2104.96**
manol-f10i	376.61	**305.58**
manol-f10bid	1381.6	**1066.62**
manol-f10id	**1279.94**	1402.43
manol-f10bidw	-	**1742.06**
velev-7pipe_q0_k	4747.46	**1796.33**
velev-11pipe_q0_k	-	-
velev-14pipe_q0_k	-	-
velev-13pipe_q0_k	-	-
velev-10pipe_q0_k	-	**5342.33**
velev-9pipe_q0_k	545.58	**129.849**
velev-15pipe_q0_k	-	-
velev-8pipe_q0_k	-	**5316.14**
velev-12pipe_k	-	-
velev-8pipe_k	-	-
velev-6pipe_k	46.57	**19.46**
velev-10pipe_k	-	-
velev-7pipe_k	-	**2721.39**
velev-9pipe_k	874.26	**232.91**
velev-13pipe_k	-	-

TABLE 1 – Zoom sur quelques familles d'instances industrielles

cient Solvers".

6 Conclusion

Dans ce papier, nous avons proposé une nouvelle approche pour l'apprentissage des clauses. Plus précisement, à chaque conflit, nous traversons le graphe d'implications séparément à partir de x et $\neg x$ (x le littéral conflit) jusqu'au Premier Unique Point d'Implication (First UIP). Au cours de cette traversée, nous dérivons une nouvelle classe de clauses Bi-Assertives qui peuvent conduire à un graphe d'implications plus compact. Plus intéressant, ces nouvelles clauses Bi-Assertives tendent à être plus courtes et induisent plus d'implications que les clauses Bi-Assertives classiques [10]. Les résultats expérimentaux montrent que l'exploitation de cette nouvelle classe de clauses Bi-Assertives permet d'améliorer les performances des solveurs issus de l'état de l'art de SAT. Notre approche bénéficie particulièrement aux instances crafted et permet des améliorations significatives sur certaines familles d'instances industrielles.

Ce travail ouvrent plusieurs perspectives intéressantes comme travaux futurs. Nous envisageons de trouver une méthode pour gérer plus finement la nouvelle base de clauses Bi-Assertives. En effet, comme le cas de la gestion de la base des clauses assertives, conserver un grand nombre de clauses Bi-Assertives peut altérer l'efficacité de la propagation unitaire, tandis que supprimer trop de clauses peut rendre l'apprentissage inefficace. Par conséquent, il faut trouver de bons critères permettant d'identifier les clauses Bi-Assertives les plus pertinentes. Enfin, nous envisageons d'exploiter le graphe d'implications proposé dans [1] pour dériver de nouvelles clauses assertives. En effet, l'addition des arcs (appelés arcs inverses) ensemble avec nos clauses Bi-Assertives pourrait conduire aux clauses assertives meilleures en termes de niveau de backjumping et de taille.

Références

[1] Gilles Audemard, Lucas Bordeaux, Youssef Hamadi, Saïd Jabbour, and Lakhdar Saïs. A generalized framework for conflict analysis. In *Proceedings of the 11th international conference on Theory and applications of satisfiability testing*, SAT'08, pages 21–27, Berlin, Heidelberg, 2008. Springer-Verlag.

[2] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem proving. *Journal of the Association for Computing Machinery*, 5 :394–397, 1962.

[3] Niklas Eén and Armin Biere. Effective preprocessing in sat through variable and clause elimination. In *Proceedings of the Eighth International Conference on Theory and Applications of Satisfiability Testing (SAT'05)*, volume 3569 of *Lecture Notes in Computer Science*, pages 61–75, St. Andrews, Scottland, June 2005. Springer.

[4] Niklas Eén and Niklas Sörenson. An extensible sat-solver. In *Proceedings of the Sixth International Conference on Theory and Applications of Satisfiability Testing (SAT'03)*, volume 2929 of *Lecture Notes in Computer Science*, pages 333–336, Santa Margherita Ligure, Italy, May 2003. Published in 2004. Springer.

[5] Carla P. Gomes, Bart Selman, and Henry Kautz. Boosting combinatorial search through randomization. In *Proceedings of the Fifteenth American National Conference on Artificial Intelligence (AAAI'97)*, pages 431–437, Madison, Wisconsin, USA, July 1998. American Association for Artificial Intelligence Press.

[6] Jinbo Huang. The effect of restarts on the efficiency of clause learning. In *Proceedings of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI'07)*, pages 2318–2323, Hyderabad, India, January 6-16 2007.

[7] H. Kautz, E. Horvitz, Y. Ruan, C. Gomes, and B. Selman. Dynamic restart policies. In *aaai02*, pages 674–682, 2002.

[8] João P. Marques-Silva and Karem A. Sakallah. Grasp—a new search algorithm for satisfiability. In *ICCAD '96 : Proceedings of the 1996 IEEE/ACM international conference on Computer-aided design*, pages 220–227, Washington, DC, USA, 1996. IEEE Computer Society.

[9] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff : engineering an efficient sat solver. In *DAC '01 : Proceedings of the 38th annual Design Automation Conference*, pages 530–535, New York, NY, USA, June 2001. ACM.

[10] Knot Pipatsrisawat and Adnan Darwiche. A new clause learning scheme for efficient unsatisfiability proofs. In *Proceedings of the Twenty-Third American National Conference on Artificial Intelligence (AAAI'08)*, pages 1481–1484, 2008.

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

[12] Knot Pipatsrisawat and Adnan Darwiche. On modern clause-learning satisfiability solvers. *Journal of Automated Reasoning*, 44(3) :277–301, 2010.

[13] Lintao Zhang, Conor F. Madigan, Matthew H. Moskewicz, and Sharad Malik. Efficient conflict driven learning in a boolean satisfiability solver. In *ICCAD '01 : Proceedings of the 2001 IEEE/ACM international conference on Computer-aided design*, pages 279–285, Piscataway, NJ, USA, November 2001. IEEE Press.

Fouille de Données pour la Compression de Formules Propositionnelles

Said Jabbour[1] **et Lakhdar Sais**[1] **et Yakoub Salhi**[1] **et Takeaki Uno**[2]

[1] CRIL - CNRS, Université d'Artois, Lens, France
[2] National Institute of Informatics, Tokyo, Japan
{jabbour, sais, salhi}@cril.fr, uno@nii.jp

Résumé

Dans cet article, nous proposons une première application des techniques de fouille de données au problème de la satisfiabilité propositionnelle. Notre approche de compression à base de fouille de donnés, a pour but de découvrir et d'exploiter les connaissances structurelles cachées pour réduire la taille des formules propositionnelles sous forme normale conjonctive (CNF). Elle combine à la fois les itemsets fréquents et l'encodage de Tseitin afin de fournir une représentation compacte des formules CNF. L'évaluation expérimentale de notre approche montre des réductions intéressantes de la taille des instances d'applications issues des dernières compétitions SAT.

Abstract

In this paper, we propose a first application of data mining techniques to propositional satisfiability. Our proposed mining based compression approach aims to discover and to exploit hidden structural knowledge for reducing the size of propositional formulae in conjunctive normal form (CNF). It combines both frequent itemset mining techniques and Tseitin's encoding for a compact representation of CNF formulae. The experimental evaluation of our approach shows interesting reductions of the sizes of many application instances taken from the last SAT competitions.

1 Introduction

La Satisfiabilité Propositionnelle (SAT), à savoir, le problème de vérifier si une formule booléenne sous forme normale conjonctive (CNF) est satisfiable ou non, est devenu une problème de base dans de nombreux domaines d'application, tels que la vérification formelle, planification et diverses nouvelles applications dérivées par les récents progrès impressionnants obtenus dans la résolution pratique de SAT.

Le succès croissant obtenu en résolvant des instances SAT du monde réel met en évidence une réelle transition à l'échelle industrielle et commerciale. Il en résulte une croissance rapide de la taille des instances CNF encodant des problèmes du monde réel. Par conséquent, la conception de nouveaux modèles efficaces pour représenter et pour résoudre les instances SAT de très grandes tailles ("Big instances") est clairement un défi important.

Récemment, une approche originale pour la compression des ensembles de clauses binaires a été proposée par J. Rintanen dans [10]. Les clauses binaires sont omniprésentes dans les formules propositionnelles qui représentent des problèmes du monde réel, allant de la vérification de modèles aux problèmes de planification en IA. Dans [10], en utilisant des variables auxiliaires, il est montré comment des graphes de contraintes contenant de grandes cliques ou bi-cliques de clauses binaires peuvent être représentés de manière plus compacte que la représentation quadratique et explicite. La principale limitation de cette approche réside dans sa restriction à des ensembles particuliers de clauses binaires dont le graphe de contraintes représente des cliques ou des bi-cliques. Ces régularités particulières peuvent être causées par la présence d' une contrainte au-plus-un sur un sous-ensemble de variables booléennes, interdisant à plus d'une variable de prendre la valeur vraie.

Dans la communauté fouille de données, plusieurs modèles et techniques pour découvrir des motifs intéressants dans les grandes bases de données ont été proposés au cours des dernières années. Le problème de fouille de motifs fréquents est bien connu et essentiel dans l'exploration de données, la découverte de connaissances et l'analyse des données. Depuis le premier article d'Agrawal [1] sur les règles d'association

et la fouille d'itemsets, un grand nombre d'ouvrages et de projets ont vu le jour, montrant l'intérêt réel de ce problème (voir [11] pour une récente étude).

Notre objectif dans ce travail est de proposer un moyen original pour compresser les formules booléennes CNF. Notre approche de compression basée sur la fouille de données a pour but de découvrir des structures cachées dans les formules CNF et de les exploiter pour réduire leur taille globale en préservant la satisfiabilité. Il s'agit de la première application des techniques de fouille de données à la satisfiabilité propositionnelle.

Dans cet article, nous présentons la première approche d'extraction de motifs pour la satisfiabilité propositionnelle. Nous montrons que les techniques de fouille de données sont très appropriées pour découvrir des motifs intéressants d'une CNF. Ces motifs sont ensuite utilisés pour réécrire la formule CNF de manière plus compacte.

2 Le problème de fouille de motifs fréquents

2.1 Notations et définitions préliminaires

Soit $\mathcal{I}$ un ensemble d'*items*. Un ensemble $I \subseteq \mathcal{I}$ est appelé *itemset*. Une *transaction* est un couple (tid, I) tel que tid est l'*identifiant de la transaction* et I est un itemset. Une *base de transactions* $\mathcal{D}$ est un ensemble fini de transactions construit sur $\mathcal{I}$ où deux transactions différentes n'ont pas le même identifiant de transaction. On dit qu'une transaction (tid, I) *supporte* un itemset J si $J \subseteq I$.

La *couverture* d'un itemset I dans une table de transactions $\mathcal{D}$ est l'ensemble d'identifiants de transactions de $\mathcal{D}$ supportant I : $\mathcal{C}(I, \mathcal{D}) = \{tid \mid (tid, J) \in \mathcal{D}$ et $I \subseteq J\}$. Le *support* d'un itemset I dans $\mathcal{D}$ est défini par : $\mathcal{S}(I, \mathcal{D}) =\mid \mathcal{C}(I, \mathcal{D}) \mid$. De plus, la *fréquence* de I dans $\mathcal{D}$ est définie par : $\mathcal{F}(I, \mathcal{D}) = \frac{\mathcal{S}(I,\mathcal{D})}{|\mathcal{D}|}$.

Soient $\mathcal{D}$ une base de transactions sur $\mathcal{I}$ et λ un seuil minimum de support. Le problème de fouille d'itemsets fréquents consiste à calculer l'ensemble suivant : $\mathcal{FIM}(\mathcal{D}, \lambda) = \{I \subseteq \mathcal{I} \mid \mathcal{S}(I, \mathcal{D}) \geq \lambda\}$.

Le problème de calculer le nombre d'itemsets fréquents est $\#P$-Difficile [6]. La classe de complexité $\#P$ correspond aux problèmes de comptage de l'ensemble associé à un problème de décision NP. Par exemple, compter le nombre de modèles satisfaisant une formule CNF est un problème $\#P$.

2.2 Itemsets Fréquents, Fermés, et Maximaux

Rappelons, à présent, deux représentations condensées d'un ensemble d'itemsets fréquents, à savoir les itemsets maximaux et les itemsets fermés.

Définition 1 (Itemset Fréquent et Maximal) *Soient $\mathcal{D}$ une base de transactions, λ un seuil minimal pour le support et $I \in \mathcal{FIM}(\mathcal{D}, \lambda)$. I est appelé maximal si pour tout $I' \supset I$, $I' \notin \mathcal{FIM}(\mathcal{D}, \lambda)$ (I' n'est pas un itemset fréquent).*

On dénote par $\mathcal{MAX}(\mathcal{D}, \lambda)$ l'ensemble de tous les itemsets fréquents et maximaux dans $\mathcal{D}$ et λ le seuil minimal pour le support.

Définition 2 (Itemset Fréquent et Fermé) *soient $\mathcal{D}$ une base de transactions, λ un seuil de support minimal et $I \in \mathcal{FIM}(\mathcal{D}, \lambda)$. I est dit fermé si pour tout $I' \supset I$, $\mathcal{C}(I, \mathcal{D}) \neq \mathcal{C}(I', \mathcal{D})$.*

On dénote par $\mathcal{CLO}(\mathcal{D}, \lambda)$ l'ensemble des itemsets fréquents et fermés dans $\mathcal{D}$ avec λ un seuil de support minimal.

On peut facilement déduire que si on dispose de tous les itemsets fréquents fermés (resp. maximaux), tous les motifs fréquents peuvent être calculées sans utiliser la base de transactions correspondante. En effet, les motifs fréquents correspondent à tous les sous-ensembles d'itemsets fréquents fermés (resp. maximaux).

Le nombre de motifs fréquents et maximaux (resp. fermés) est significativement plus faible que le nombre de motifs fréquents. Néanmoins, ce nombre n'est pas toujours polynomial en la taille de la base de transactions [14]. En particulier, le problème du comptage du nombre de motifs fréquents maximaux est # P-complet (voir aussi [14]).

3 De la forme CNF à la base de transactions

Nous présentons d'abord le problème de satisfiabilité et quelques notations nécessaires. Nous considérons que les formules propositionnelles sous forme normale conjonctive (CNF). Une *CNF* Φ est une conjonction de clauses, où une *clause* est une disjonction de littéraux. Un *littéral* est une variable positive (p) ou négative ($\neg p$). Les deux littéraux p et $\neg p$ sont dit *complémentaires*. La taille de la CNF Φ est défini comme $|\Phi| = \sum_{c \in \Phi} |c|$, où $|c|$ est égal au nombre de littéraux dans c. Une clause *unitaire* est une clause contenant un seul littéral (appelé *littéral unitaire*), alors qu'une clause binaire contient exactement deux littéraux. Une formule qui ne contient que des clauses binaires est appelé formule *2-CNF*. Une *clause vide*, notée $\perp$, est interprétée comme fausse (insatisfiable), alors qu'une

CNF vide, notée $\top$, est interprétée comme vraie (satisfiable).

Soient c_1 et c_2 deux clauses de Φ. On dit que c_1 *subsume* c_2 si $c_1 \subseteq c_2$. Si c_1 subsume c_2, alors la clause c_2 peut être supprimée de Φ en préservant la satisfiabilité. Une formule Φ est *fermée par subsumption* si $\forall c \in \Phi,\ \nexists c' \in \Phi$ tel que $c \neq c'$ et c' subsume c. On désigne par Φ^s, la formule obtenue à partir de Φ en éliminant toutes les clauses subsumées.

Nous notons $\bar{l}$ le *complémentaire* d'un littéral de l. Plus précisément, si $l = p$, alors $\bar{l}$ est $\neg p$ et si $l = \neg p$, alors $\bar{l}$ est p.

On note $\mathcal{V}_\Phi$ l'ensemble des variables propositionnelles apparaissant dans Φ, tandis que l'ensemble des littéraux de Φ est défini comme $\mathcal{L}_\Phi$.

Un *interprétation* $\mathcal{B}$ d'une formule propositionnelle Φ est une fonction qui associe une valeur $\mathcal{B}(p) \in \{0, 1\}$ (0 correspond à $faux$ et 1 à $vrai$) pour les variables $p \in \mathcal{V}_\Phi$. Un *modèle* d'une formule Φ est une interprétation $\mathcal{B}$ satisfaisant la formule : $\mathcal{B}(\Phi) = 1$. Le *problème SAT* consiste à décider si une formule CNF donnée admet un modèle ou non. On définit $\Phi|_x$ comme étant la formule obtenue à partie de Φ en affectant x à vrai. Formellement $\Phi|_x = \{c \mid c \in \Phi, \{x, \neg x\} \cap c = \emptyset\} \cup \{c \backslash \{\neg x\} \mid c \in \Phi, \neg x \in c\}$. Φ^* denote la formule Φ *fermée par propagation unitaire*, définie récursivement comme suit :
(1) $\Phi^* = \Phi$ si Φ ne contient pas de clause unitaire,
(2) $\Phi^* = \bot$ si Φ contient deux clauses unitaires $\{x\}$ et $\{\neg x\}$,
(3) autrement, $\Phi^* = (\Phi|_x)^*$ où x est un littéral apparaissant dans une clause unitaire de Φ.

Une formule CNF peut être considérée comme une base de transactions, appelée une base CNF, où les items correspondent aux littéraux et les transactions aux clauses. Notons que les littéraux complémentaires correspondent à deux items différents.

Définition 3 (de CNF à $\mathcal{D}$) *Soit $\Phi = \bigwedge_{1 \leq i \leq n} c_i$ une formule CNF. L'ensemble des items $\mathcal{I} = \mathcal{L}_\Phi$ et la base de transactions associée à Φ est définie comme $\mathcal{D}^c_\Phi = \{(tid_i, c_i) | 1 \leq i \leq n\}$*

Par exemple, la formule CNF $(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (x_1 \vee \neg x_2 \vee x_4) \wedge x_1 \wedge (x_3 \vee \neg x_4)$ correspond à la base de transactions suivante :

tid	itemset
1	$x_1, \neg x_2, \neg x_3$
2	$x_1, \neg x_2, x_4$
3	x_1
4	$x_3, \neg x_4$

Dans ce contexte, un itemset fréquent correspond à un ensemble de littéraux fréquents : le nombre de clauses contenant ces littéraux est supérieur ou égal au seuil minimal pour le support. Par exemple, si nous fixons le seuil minimal λ à 2, nous obtenons $\{x_1, \neg x_2\}$ comme motif fréquent dans la base de transactions précédente. L'ensemble des motifs fréquents maximaux est le plus petit ensemble de motifs fréquents de littéraux où chaque ensemble fréquent de littéraux est inclus dans au moins un de ses éléments. Par exemple, l'unique motif fréquent maximale dans l'exemple précédent est $\{x_1, \neg x_2\}$ ($\lambda = 2$). En outre, l'ensemble des motifs fréquents fermés est le plus petit ensemble de motifs fréquents de littéraux où chaque motif fréquent est inclus dans au moins un de ses éléments *ayant le même support*. Par exemple, l'ensemble des motifs fréquents fermés est $\{\{x_1, \neg x_2\}, \{x_1\}\}$.

Dans la définition d'une base de transactions, nous n'avons pas besoin que l'ensemble des items d'une transaction soit unique. En effet, deux transactions différentes peuvent avoir le même ensemble d'éléments mais avec des identifiants différents.

4 Compression de CNF fondée sur la fouille de données

Dans cette section, nous décrivons notre approche de compression fondée sur la fouille de données, appelée `Mining4SAT`, pour réduire la taille d'une formule CNF. L'idée principale consiste à rechercher des ensembles fréquents de littéraux (sous-clauses) et les remplacer par de nouvelles variables en utilisant l'encodage de Tseitin [12].

4.1 Transformation de Tseitin

La transformation de Tseitin consiste à introduire de nouvelles variables pour représenter des sous-formules. Par exemple, pour deux variables données a et b d'une formule booléenne, et v une nouvelle variable. On peut ajouter la définition $v \leftrightarrow a \vee b$ (appelé extension) à la formule tout en préservant la satisfiabilité. Le principe de l'extension de Tseitin est à la base de la transformation linéaire de formules booléennes générales en CNF. Deux décennies plus tard, après un article fondateur de Tseitin, Plaisted et Greenbaum ont présenté une traduction en CNF plus améliorée qui produit essentiellement un sous-ensemble de la représentation de Tseitin [8]. Ils ont remarqué que par le suivi des polarités de sous-formules, on peut éliminer une grande partie de la traduction de Tseitin. Dans la suite, nous utilisons l'approche de Plaisted et Greenbaum. Plus précisément, comme la disjonction $a \vee b$ est une sous-clause de polarité positive, il suffit d'ajouter la formule $v \rightarrow a \vee b$ i.e., la clause $(\neg v \vee a \vee b)$

Considérons la formule DNF suivante (une disjonction de conjonctions) :

$$(x_1 \wedge \cdots \wedge x_l) \vee (y_1 \wedge \cdots \wedge y_m) \vee (z_1 \wedge \cdots \wedge z_n)$$

Une manière naïve de convertir une telle formule en CNF consiste à appliquer la distributivité de la disjonction sur la conjonction.

$$(x_1 \vee y_1 \vee z_1) \wedge (x_1 \vee y_1 \vee z_2) \wedge \cdots \wedge (x_l \vee y_m \vee z_n)$$

Une telle approche naïve est clairement exponentielle dans le pire des cas. Dans la transformation de Tseitin, des variables auxiliaires sont introduites pour empêcher une telle explosion combinatoire, principalement causée par la distributivité de la disjonction sur la conjonction et vice versa. Avec les variables supplémentaires, la formule CNF obtenue est linéaire en la taille de la formule d'origine. Cependant uniquement l'équivalence pour la satisfiabilité est préservée :

$$(t_1 \vee t_2 \vee t_3) \wedge (t_1 \rightarrow (x_1 \wedge \cdots \wedge x_l)) \wedge (t_2 \rightarrow (y_1 \wedge \cdots \wedge y_m))$$

$$\wedge (t_3 \rightarrow (z_1 \wedge \cdots \wedge z_n))$$

4.2 Réduction de la taille des formules CNF

Décrivons plus en détail comment les techniques de fouille d'itemsets peuvent être combinées avec le principe de Tseitin pour compresser les formules CNF.

Afin d'illustrer notre approche fondée sur l'exploitation des techniques de fouille pour la compression, nous considérons la formule CNF Φ suivante :

$$(x_1 \vee \cdots \vee x_n \vee \alpha_1) \wedge \cdots \wedge (x_1 \vee \cdots \vee x_n \vee \alpha_k)$$

telle que $n \geq 2$, $k > \frac{n+1}{n-1}$ et $\alpha_1, \ldots, \alpha_k$ sont des clauses. Comme on peut le remarquer, la sous-clause $(x_1 \vee \ldots, \vee x_n)$ apparait dans chacune des clauses de Φ. En utilisant la transformation de Tseitin, on peut réécrire Φ comme suit :

$$(y \vee \alpha_1) \wedge \cdots \wedge (y \vee \alpha_k) \wedge (x_1 \vee \cdots \vee x_n \vee \neg y)$$

où y est une variable additionnelle. En effet, $n \times k$ littéraux sont remplacés par $k+n+1$ littéraux conduisant à un gain par rapport au nombre de littéraux de $(n \times k) - (n + k + 1)$.

Maintenant, si l'on considère la base CNF correspondant à Φ, $\{x_1, \ldots, x_n\}$ est un itemset fréquent où le seuil minimal pour le support est supérieur ou égal à k. Il est facile de voir que, pour réduire le nombre de littéraux, n doit être supérieur ou égal à 2. En effet, si $n < 2$, il n'y a pas de réduction du nombre de littéraux, au contraire, leur nombre croît. En ce qui concerne la valeur de k, on peut aussi voir qu'une telle transformation n'est intéressante que lorsque k $> \frac{n+1}{n-1}$. Ainsi, il existe trois cas : si $n = 2$, alors $k \geq 4$, sinon si $n = 3$, alors $k \geq 3$, autrement $k \geq 2$. Par conséquent, le nombre de littéraux est toujours réduit lorsque $k \geq 4$.

De toute évidence, une interprétation booléenne est un modèle de la formule obtenue après réduction si et seulement si il est un modèle de Φ.

Dans l'exemple précédent, nous montrons comment le problème de trouver les motifs fréquents peut être utilisé pour réduire la taille d'une formule CNF. On peut voir que, en général, il est plus intéressant d'envisager une représentation condensée des motifs fréquents (fermés et maximaux) pour réduire le nombre de littéraux. En effet, en utilisant une représentation condensée, nous considérons tous les motifs fréquents et le nombre de variables propositionnelles auxiliaires et les nouvelles clauses (dans notre exemple, y et $(x_1 \vee \cdots \vee x_n \vee \neg y)$) introduite est inférieure à celles en utilisant tous les motifs fréquents. Par exemple, dans la formule précédente, il n'est pas intéressant d'introduire une nouvelle variable propositionnelle pour chaque sous-ensemble de $\{x_1, \ldots, x_n\}$.

Exemple 1 *Considérons la formula Φ contenant les 4 clauses suivantes :*

$$\begin{array}{lcl} & & \overline{} \\ \neg x_0 \vee x_1 & \vee & \mid x_4 \vee x_5 \vee x_6 \mid \\ x_3 & \vee & \mid x_4 \vee x_5 \vee x_6 \mid \\ \neg x_1 \vee x_2 & \vee & \mid x_4 \vee x_5 \vee x_6 \mid \\ \neg x_2 \vee x_3 & \vee & \mid x_4 \vee x_5 \vee x_6 \mid \\ & & \underline{} \end{array}$$

Supposons que le seuil de support minimal est inférieur à 4, alors la sous-clause $(x_4 \vee x_5 \vee x_6)$ est fréquente. En utilisant notre approche, la formule Φ peut être réécrite sous la forme :

$$\begin{array}{lcl} & & \overline{\phantom{\mathbf{y}}} \\ \neg x_0 \vee x_1 & \vee & \mid \mathbf{y} \mid \\ x_3 & \vee & \mid \mathbf{y} \mid \\ \neg x_1 \vee x_2 & \vee & \mid \mathbf{y} \mid \\ \neg x_2 \vee x_3 & \vee & \mid \mathbf{y} \mid \\ & & \underline{\phantom{\mathbf{y}}} \\ \neg \mathbf{y} & \vee & x_4 \vee x_5 \vee x_6 \end{array}$$

Dans cet exemple simple, la formule originale contient 27 littéraux, tandis que la nouvelle formule contient uniquement 23 littéraux.

Fermés vs. Maximaux Dans la Section 2.2, nous introduisons deux représentations condensées des itemets fréquents : fermés et maximaux. La question est : quelle est la meilleure représentation ? Nous savons que l'ensemble des motifs fréquents maximaux est inclus dans celui des fermés. Ainsi, un petit nombre de nouvelles variables et de nouvelles clauses sont introduites en utilisant des motifs fréquents maximaux. Cependant, il y'a des cas où l'utilisation des motifs fréquents fermés est plus appropriée. Par exemple, considérons la formule suivante :

$$
\begin{array}{ll}
(x_1 \vee \ldots \vee x_k \vee \ldots \vee x_n \vee \alpha_1) & \wedge \\
\vdots & \vdots \\
(x_1 \vee \ldots \vee x_k \vee \ldots \vee x_n \vee \alpha_m) & \wedge \\
(x_1 \vee \ldots \vee x_k \vee \beta_1) & \wedge \\
\vdots & \vdots \\
(x_1 \vee \ldots \vee x_k \vee \beta_{m'}) &
\end{array}
$$

telle que $k \geq 2$, $m, m' \geq 4$ et $n > k$. Nous supposons que les motifs fréquents sont les sous-ensembles de $\{x_1, \ldots, x_n\}$. Par conséquent, $\{x_1, \ldots, x_n\}$ est l'unique itemset maximal et les itemsets fermés sont $\{x_1, \ldots, x_n\}$ et $\{x_1, \ldots, x_k\}$.

Commençons par remplacer l'itemset fréquent et fermé $\{x_1, \ldots, x_n\}$ dans la réduction du nombre de littéraux :

$$
\begin{array}{ll}
(y \vee \alpha_1) \wedge \ldots \wedge (y \vee \alpha_m) & \wedge \\
(x_1 \vee \ldots \vee x_k \vee \beta_1) & \wedge \\
\vdots & \vdots \\
(x_1 \vee \ldots \vee x_k \vee \beta_{m'}) & \wedge \\
(x_1 \vee \ldots \vee x_n \vee \neg y) &
\end{array}
$$

Maintenant, en utilisant $\{x_1, \ldots, x_k\}$, nous obtenons la formule suivante :

$$
\begin{array}{ll}
(y \vee \alpha_1) \wedge \ldots \wedge (y \vee \alpha_m) & \wedge \\
(z \vee \beta_1) \wedge \ldots \wedge (z \vee \beta_{m'}) & \wedge \\
(z \vee x_{k+1} \vee \ldots \vee x_n \vee \neg y) & \wedge \\
(x_1 \vee \ldots \vee x_k \vee \neg z) &
\end{array}
$$

Dans cet exemple, il est plus intéressant de considérer les motifs fréquents fermés dans notre approche Mining4SAT.

En fait, un motif (fermé) fréquent I et l'un de ses sous-ensembles I' (qui peut être fermé) sont à la fois intéressants si $\mathcal{S}(I') - \mathcal{S}(I) > \frac{|I'|1}{|I'|-1} - 1$. En effet, si nous appliquons notre transformation en utilisant I, puis le support de I' dans la formule qui en résulte est égal à $\mathcal{S}(I') - \mathcal{S}(I) + 1$, et nous savons que I' est intéressant dans la formule résultante si son support est supérieur à $\frac{|I'|+1}{|I'|-1}$.

Chevauchement Soit Φ un ensemble d'itemsets. Deux itemsets I et I' de Φ se chevauchent si $I \cap I' \neq \emptyset$. De plus, I et I' sont dans la même *classe de chevauchement* s'il existe k itemsets $I_1, \ldots, I_k$ de Φ tels que $I = I_1, I_k = I'$ et pour tout $1 \leq i \leq k-1$, I_i et I_{i+1} se chevauchent.

Dans notre transformation, on peut avoir quelques problèmes lorsque deux motifs fréquents se chevauchent.

Par example, si $\{x_1, x_2, x_3\}$ et $\{x_2, x_3, x_4\}$ sont deux itemsets fréquents (3 est le seuil minimum pour le support) tel que $\mathcal{S}(\{x_1, x_2, x_3\}) = 3$, $\mathcal{S}(\{x_2, x_3, x_4\}) = 3$ et $\mathcal{S}(\{x_1, x_2, x_3, x_4\}) = 2$, alors si nous appliquons notre transformation en utilisant $\{x_1, x_2, x_3\}$, alors le support de $\{x_2, x_3, x_4\}$ est égal à 2 (non fréquent) dans la formule résultante et vice versa. Ainsi, nous ne pouvons pas utiliser les deux dans la transformation.

Notons que la notion de chevauchement peut être vue comme une généralisation de celle de sous-ensemble. Soient I et I' deux motifs fréquents tels qu'ils se chevauchent. Les deux motifs sont intéressants dans notre transformation si :

1. $\mathcal{S}(I) - \mathcal{S}(I \cup I') > \frac{|I|+1}{|I|-1} - 1$ où $\mathcal{S}(I') - \mathcal{S}(I \cup I') > \frac{|I'|+1}{|I'|-1} - 1$. Cela provient du fait que si l'on applique la transformation en utilisant I (resp. I'), alors le support de I' (resp. I) est égal à $\mathcal{S}(I') - \mathcal{S}(I \cup I') + 1$ (resp. $\mathcal{S}(I) - \mathcal{S}(I \cup I') + 1$).

2. $|I \backslash I'| \geq k$ (resp. $|I' \backslash I| \geq k$) tel que $k = 2$ si $\mathcal{S}(I) \geq 4$ (resp. $\mathcal{S}(I') \geq 4$), $k = 3$ si $\mathcal{S}(I) = 3$ (resp. $\mathcal{S}(I') = 3$), $k = 4$ autrement. En effet, dans les cas précédents, $I \backslash I'$ (resp. $I' \backslash I$) peut être utilisé dans notre transformation.

L'algorithme Mining4SAT Nous décrivons maintenant notre algorithme de compression, appelé `Mining4SAT`, en utilisant l'ensemble des motifs fréquents fermés. Notons que la transformation optimale en utilisant l'ensemble de tous les motifs fréquents fermés peut être obtenue par une transformation optimale en utilisant séparément les classes de chevauchement de cet ensemble. En fait, puisque deux classes de chevauchement distinctes ne partagent pas de littéraux, la réduction appliquée pour une formule donnée à l'aide des éléments d'une classe de chevauchement n'affecte pas les supports des éléments des autres classes. En outre, on peut facilement calculer l'ensemble de toutes les classes de chevauchement de l'ensemble des motifs fréquents fermés : soit $G = (V, E)$ un graphe non orienté de telle sorte que V est l'ensemble des motifs fréquents fermés et (I_1, I_2) est une arête de G si et seulement si I_1 et I_2 se chevauchent ; C est une classe de chevauchement si et seulement si elle correspond à l'ensemble des sommets d'une composante connexe de G, ce qui n'est pas inclus dans aucune autre composante connexe de G. Pour cette raison, nous nous limitons ici aux réductions qui peuvent être obtenues en utilisant une seule classe de chevauchement. L'ensemble du processus de réduction de la taille peut être réalisé en effectuant une itération sur toutes les classes de chevauchement.

Soit I un itemset fréquent fermé, on note $\alpha(I)$ la valeur $\mathcal{S}(I) \times (|I| - 1) - |I| - 1$ qui correspond au nombre de littéraux réduits par l'application de notre transformation avec I sur une formule CNF.

L'algorithme 1 prend en entrée une formule CNF ϕ et une classe de chevauchement C, et retourne ϕ après l'application de l'algorithme de réduction de la taille. On effectue des itérations jusqu'à ce qu'il n'y

Algorithm 1 Réduction de la taille

Require: Une formule ϕ, une classe de chevauchement d'itemsets fréquents et fermés C

1: **while** $C \neq \emptyset$ **do**
2: $I \leftarrow MostInterstingElment(C)$;
3: $replace(\phi, I, x)$;
4: $Add(\phi, I, x)$:
5: $remove(C, I)$;
6: $replaceSubset(C, I, x)$;
7: $removeUninterestingElements(C)$;
8: $updateSupports(C)$;
9: **end while**
10: **return** ϕ

ait aucun élément dans C. À chaque itération, on sélectionne d'abord l'un des éléments les plus intéressants dans C (ligne 2) : un élément I de C tel que aucun élément $I' \in C$ satisfaisant $\alpha(I') > \alpha(I)$. Notez que cet élément n'est pas nécessairement unique dans C. Cette instruction signifie que l'algorithme 1 est un algorithme glouton, car il fait un choix localement optimal à chaque itération. Ensuite, il applique notre transformation en utilisant $I = \{y_1, \ldots, y_n\}$: il remplace les occurrences de I avec une nouvelle variable propositionnelle x (ligne 3) ; et il ajoute la clause $y_1 \vee \ldots \vee y_n \vee \neg x$ à ϕ (ligne 4). Il supprime I de C (ligne 5) et remplace I dans les autres éléments de C par x (ligne 6). La prochaine instruction (ligne 7) consiste à retirer les éléments de C qui pourraient augmenter le nombre de littéraux : les éléments qui se chevauchent avec I et qui ne sont pas inclus dans I. Comme expliqué précédemment, un élément de C qui se chevauche avec I n'augmente pas nécessairement le nombre de littéraux. Ainsi, par la suppression d'éléments de C parce qu'ils se chevauchent avec I, notre algorithme peut supprimer des motifs fréquents fermés diminuant le nombre de littéraux. Une solution partielle à ce problème consiste à recalculer les motifs fréquents fermés dans la formule retournée par l'algorithme 1. La dernière instruction dans la boucle while (ligne 8) consiste à mettre à jour les supports des éléments restants dans C : le support d'un élément I' restant dans C change lorsqu'il est inclus dans I et son nouveau support est égal à $\mathcal{S}(I') - \mathcal{S}(I) + 1$. Cette instruction supprime également tous les éléments de C devenant sans intérêt en raison des nouveaux supports et tailles.

5 Application : Une représentation Compacte d'un ensemble de clauses binaires

Les clauses binaires (formule 2-CNF) sont omniprésentes dans les formule CNF encodant des problèmes du monde réel. Certaines d'entre elles contiennent plus de 90 % de clauses binaires. Une des raisons principales est que l'encodage de plusieurs types de contraintes en CNF conduit à de grands ensembles de clauses binaires. Par exemple, l'expression que les variables $x_1, \ldots, x_n$ doivent prendre des valeurs différentes dans $\{v_1, \ldots, v_m\}$ conduit à $n^2 \times m^2$ clauses binaires (cliques de clauses binaires) avec un encodage naïf. Pour $n = 100$ et $m = 10$, nous obtenons environ un million de clauses binaires ou 10 mégaoctets si chaque clause binaire est codée avec `10 octets`. Un autre exemple donné par Rintanen dans [10], concerne l'encodage de problèmes de planification en SAT et en particulier de certains invariants tels que celui exprimant qu'un objet ne peut pas être à deux endroits en même temps. Pour n variables d'état il y a $\frac{n \times (n-1)}{2}$ invariants qui sont des clauses binaires. Dans le cas d'un problème de planification avec $n = 5000$ variables d'état et une formule qui code pour la recherche de plans de 100 points, si ce n'est que l'une des variables d'état qui est vrai à un moment donné, la taille totale de l'ensemble de clauses binaires est d'environ 12 *gigaoctets*.

5.1 Compression d'un ensemble arbitraire de clauses binaires

Dans cette section, nous montrons comment notre approche fondée sur la fouille de données peut être utilisée pour obtenir une représentation compacte d'un ensemble arbitraire de clauses binaires. Ensuite, nous considérons deux cas particuliers intéressants correspondant à des ensembles de clauses binaires représentant soit une clique ou une bi-clique. Il est important de noter que, dans [10], les auteurs ont étudié ces seuls cas particuliers.

Afin de réaliser des réductions importantes en nombre de littéraux, mais aussi en nombre de clauses, nous proposons une approche en quatre étapes pour la compression d'un ensemble de clauses binaires. Dans la première étape, nous réécrivons l'ensemble de clauses binaires en utilisant une autre représentation plus appropriée. Deuxièmement, à partir de cette représentation intermédiaire, nous construisons une nouvelle base de transactions. Ensuite, nous cherchons des itemsets fréquents et fermés. Enfin, nous appliquons l'algorithme de compression obtenu par une légère modification de l'algorithme 1 sur l'ensemble de clauses binaires.

Nous présentons d'abord une représentation plus pratique et équivalente d'un ensemble de clauses binaires.

Définition 4 (B-implication) *Une B-implication est une formule booléenne de la forme : $x \vee \beta(x)$ tel que $\beta(x)$ est une conjonction de littéraux.*

Soit S l'ensemble suivant de clauses binaires : $\{(x \vee y_1), \ldots, (x \vee y_k)\}$, avec $k \geq 1$. Notons que la B-implication $B(S) = x \vee (y_1 \wedge \cdots \wedge y_k)$ est équivalente

Algorithm 2 B-implications

Require: Une formule Φ, un ordre total $\preceq$ sur $\mathcal{L}(\Phi)$
1: $\mathcal{C}(x) = \emptyset,\ \forall x \in \mathcal{L}(\Phi)$
2: **for** $c = (x \vee y) \in \Phi$ **do**
3: **if** $x \prec y$ **then**
4: $\mathcal{C}(x) \leftarrow \mathcal{C}(x) \cup \{y\}$
5: **else**
6: $\mathcal{C}(y) \leftarrow \mathcal{C}(y) \cup \{x\}$
7: **end if**
8: **end for**
9: **for** $x \in \mathcal{L}(\Phi)$ **do**
10: **if** $\mathcal{C}(x) = \{y\}$ *and* $|\mathcal{C}(y)| > 1$ **then**
11: $\mathcal{C}(y) \leftarrow \mathcal{C}(y) \cup \{x\}$
12: $\mathcal{C}(x) \leftarrow \emptyset$
13: **end if**
14: **end for**
15: **return** $\{[x \vee (\bigwedge_{y \in \mathcal{C}(x)} y)] | x \in \mathcal{L}(\Phi) \mid \mathcal{C}(x) \neq \emptyset\}$

à la conjonction des éléments de S. Ainsi, chaque formule 2-CNF Φ peut être transformée en un ensemble de B-implications, noté $B_{[\vee(\wedge)]}(\Phi)$.

La formule originale Φ peut être obtenue à partir de $B_{[\vee(\wedge)]}(\Phi)$ en distribuant $\vee$ sur $\wedge$. Cependant, Il existe plusieurs façons de réécrire une 2-CNF sous la forme d'une conjonction de B-implications. Une approche naïve consiste simplement à fixer complètement l'ordre de la relation sur $\mathcal{L}_\Phi$.

Nous définissons un ordre total sur $\mathcal{L}_\Phi$. Soit f une bijection de $\mathcal{L}_\Phi$ vers $\{1 \ldots |\mathcal{L}_\Phi|\}$. Un littéral x est inférieur à un littéral y, noté $x \preceq y$, si et seulement si $f(x) \leq f(y)$. De cette façon, chaque littéral de Φ est associé à un entier naturel unique. La relation $\preceq$ est un ordre total. Maintenant, en utilisant cet ordre, nous obtenons une façon unique de réécrire une formule 2-CNF comme un ensemble de B-implications. Soient Φ une formule 2-CNF et $(x \vee y) \in \Phi$, nous ajoutons de manière conjonctive y (respectivement x) à $\beta(x)$ (respectivement $\beta(y)$), si $x \prec y$ (respectivement $y \prec x$).

L'algorithme 2 consiste à calculer l'ensemble des B-implications associées à une formule 2-CNF Φ. Il nécessite une formule 2-CNF Φ et un ordre total $\preceq$ sur $\mathcal{L}(\Phi)$ comme entrée, et fournit un ensemble de B-implications $B_{[\vee(\wedge)]}(\Phi)$ en sortie. Dans la ligne 1, on initialise $\mathcal{C}(x)$ par l'ensemble vide pour tous les littéraux de Φ. Suite à notre relation d'ordre, dans les lignes $2-9$, nous construisons l'ensemble $\beta(x)$ pour chaque littéral $x \in \mathcal{L}_\Phi$. En effet, pour chaque clause binaire $(x \vee y) \in \Phi$, y est ajouté à $\mathcal{C}(x)$ (respectivement x est ajouté à $\mathcal{C}(y)$) si $x \prec y$ (respectivement $y \prec x$).

Dans les lignes $10-14$, si selon l'ordre choisi, nous avons $x \prec y$ et $\beta(x)$ ne contient qu'un seul littéral, alors nous essayons d'améliorer la compression de l'ensemble des B-implications $B_{[\vee(\wedge)]}(\Phi)$. Dans ce cas, nous ajoutons x à l'ensemble $\mathcal{C}(y)$, que s'il contient au moins un littéral, et nous initialisons $\mathcal{C}(x)$ à vide. Dans la ligne 16, nous retournons les B-implications de la forme $[x \vee \beta(x)]$ telles que $\mathcal{C}(x)$ n'est pas vide.

En supposant que l'ajout d'un élément à un ensemble peut être réalisé en temps constant, la complexité au pire des cas de l'algorithme 2 est en $\mathcal{O}(|\Phi| + |\mathcal{L}(\Phi)|)$.

Maintenant, nous expliquons comment une base de transactions est associée à une formule 2-CNF, en utilisant un ensemble de B-implications que nous considérons comme une représentation intermédiaire.

Définition 5 (De 2-CNF à $\mathcal{D}$) *Soit Φ une formule 2-CNF. La base de transactions associée à Φ est défini comme $\mathcal{D}^b_\Phi = \{(tid_{x_i}, \beta_i) | x_i \vee \beta_i \in B_{[\vee(\wedge)]}(\Phi)\}$.*

L'algorithme Mining4Binary Nous allons décrire notre approche de compression d'une formule 2-CNF Φ, appelée `Mining4Binary` (pour réduire la taille d'un ensemble de clauses binaires). Tout d'abord, après la réécriture de Φ comme $B_{[\vee(\wedge)]}(\Phi)$, nous construisons la base de transactions $\mathcal{D}^b_\Phi$. Ensuite, l'ensemble des motifs fréquents fermés et ses classes de chevauchement associés $\mathcal{C}$ sont calculés. La dernière étape a pour objectif de réduire la taille de la formule 2-CNF Φ en utilisant une version légèrement modifiée de l'algorithme 1. Nous avons seulement besoin d'ajouter deux modifications. Premièrement, notre algorithme de compression des 2-CNF prend en entrée $B_{[\vee(\wedge)]}(\Phi)$ et renvoie un ensemble condensé des B-implications . Deuxièmement, pour un itemset $I = \{y_1, \ldots, y_n\}$, à la ligne 4 de l'algorithme 1, nous introduisons une nouvelle variable x et nous ajoutons une B-implication $[\neg x \vee (y_1 \wedge y_2 \wedge \cdots \wedge y_n)]$ à $B_{[\vee(\wedge)]}(\Phi)$. Cet algorithme modifié retourne un ensemble de B-implications condensé. La dernière étape est une traduction triviale des B-implications obtenues en 2-CNF.

Il est évident que le taux de compression dépend de l'ordre choisi. En effet, la représentation intermédiaire (B-implications) est construite selon un ordre total, et la base de transactions dépend de cette représentation intermédiaire.

Exemple 2 *Considérons la formule 2-CNF Φ suivante :*

$$\begin{aligned}\Phi = \\ &(x_1 \vee x_2) \wedge (x_1 \vee x_3) \wedge (x_1 \vee x_4) \wedge (x_1 \vee x_5) \wedge \\ &(x_1 \vee x_6) \wedge (x_1 \vee x_7) \wedge (x_2 \vee x_3) \wedge (x_2 \vee x_4) \wedge \\ &(x_2 \vee x_5) \wedge (x_2 \vee x_6) \wedge (x_2 \vee x_7) \wedge (x_3 \vee x_4) \wedge \\ &(x_3 \vee x_6) \wedge (x_3 \vee x_7) \wedge (x_3 \vee x_5) \wedge (x_4 \vee x_5) \wedge \\ &(x_4 \vee x_6) \wedge (x_4 \vee x_7) \wedge (x_5 \vee x_6) \wedge (x_5 \vee x_7) \wedge \\ &(x_6 \vee x_7)\end{aligned}$$

En utilisant l'ordre total $x_1 \prec \ldots \prec x_7$ sur $\mathcal{L}_\Phi$, nous pouvons réécrire Φ sous la forme de l'ensemble des B-implications $B^1_{[\vee(\wedge)]}(\Phi)$:

$$\begin{aligned}B^1_{\vee[\wedge]}(\Phi) = \quad &\{[x_1 \vee (x_2 \wedge x_3 \wedge x_4 \wedge x_5 \wedge x_6 \wedge x_7)], \\ &[x_2 \vee (x_3 \wedge x_4 \wedge x_5 \wedge x_6 \wedge x_7)], \\ &[x_3 \vee (x_4 \wedge x_5 \wedge x_6 \wedge x_7)], \\ &[x_5 \vee (x_6 \wedge x_7)], \\ &[x_6 \vee (x_7)]\}\end{aligned}$$

La base de transaction $\mathcal{D}^b_\Phi$ est construite à partir de $B^1_{[\vee(\wedge)]}(\Phi)$ en considérant uniquement $\beta(x_1), \ldots, \beta(x_6)$:

tid	$itemset$					
tid_{x_1}	x_2	x_3	x_4	x_5	x_6	x_7
tid_{x_2}		x_3	x_4	x_5	x_6	x_7
tid_{x_3}			x_4	x_5	x_6	x_7
tid_{x_4}				x_5	x_6	x_7
tid_{x_5}					x_6	x_7
tid_{x_6}						x_7

Le processus de fouille des itemsets se fait sur la partie conjonctive de $B^1_{\vee[\wedge]}(\Phi)$ représentée dans la base de transactions. En fixant le seuil de support minimum à 4, nous obtenons comme motifs fréquents $\{x_5, x_6, x_7\}$. En utilisant l'algorithme de compression des 2-CNF décrit ci-dessus, nous pouvons réécrire $B^1_{[\vee(\wedge)]}(\Phi)$ sous la forme $B^2_{[\vee(\wedge)]}(\Phi)$:

$$\begin{aligned} B^2_{\vee[\wedge]}(\Phi) = \quad & \{[x_1 \vee (x_2 \wedge x_3 \wedge y)] \ , \\ & [x_2 \vee (x_3 \wedge x_4 \wedge y)] \ , \\ & [x_3 \vee (x_4 \wedge y)] \ , \\ & [x_5 \vee (x_6 \wedge x_7)] \ , \\ & [x_6 \vee (x_7)] \ , \\ & [\neg y \vee (x_5 \wedge x_6 \wedge x_7)]\} \end{aligned}$$

Finalement un simple encodage de $B^2_{[\vee(\wedge)]}(\Phi)$ sous forme CNF conduit à la formule 2-CNF compressée suivante :

$$\begin{aligned} & \Phi = \\ & (x_1 \vee x_2) \wedge (x_1 \vee x_3) \wedge (x_1 \vee x_4) \wedge (x_2 \vee x_3) \wedge \\ & (x_2 \vee x_4) \wedge (x_3 \vee x_4) \wedge (x_5 \vee x_6) \wedge (x_5 \vee x_7) \wedge \\ & (x_6 \vee x_7) \wedge \\ & (x_1 \vee y) \wedge (x_2 \vee y) \wedge (x_3 \vee y) \wedge (x_4 \vee y) \wedge \\ & (x_5 \vee \neg y) \wedge (x_6 \vee \neg y) \wedge (x_7 \vee \neg y) \end{aligned}$$

La substitution de l'itemset $\{x_5, x_6, x_7\}$ permet de réduire la taille de la formule 2-CNF Φ. En effet, la 2-CNF résultante contient 5 clauses binaires en moins.

5.2 Cas particulier de (Bi-)cliques d'un ensemble de clauses binaires

Dans [10], J. Rintanen a abordé le problème de la représentation de grands ensembles de clauses binaires de manière compacte. Il a notamment montré que les graphes de contraintes issues d'applications réelles (par exemple de la planification en AI) contiennent de grandes cliques ou bi-cliques de clauses binaires. Une bi-clique impliquant les deux ensembles de littéraux $\mathcal{X} = \{x_1, x_2, \ldots, x_n\}$ et $\mathcal{Y} = \{y_1, y_2, \ldots, y_m\}$ exprime la formule propositionnelle $\Phi = (x_1 \wedge x_2 \wedge \cdots \wedge x_n) \vee (y_1 \wedge y_2 \wedge \cdots \wedge y_m)$, alors qu'une clique impliquant les littéraux $\mathcal{X} = \{x_1, x_2, \ldots, x_n\}$ exprime qu'au plus un littéral de $\mathcal{X}$ est à $faux$.

Bi-cliques de Clauses Binaires Expliquons comment une bi-clique peut être compressée avec la méthode Mining4Binary. Soit $\Phi = [(x_1 \vee y_1) \wedge (x_1 \vee y_2) \vee \cdots \vee (x_1 \vee y_m)] \ldots [(x_n \vee y_1) \wedge (x_n \vee y_2) \vee \cdots \vee (x_n \vee y_m)]$ une bi-clique de $n \times m$ de clauses binaires (voir Figure 1). Compte tenu de la relation d'ordre totale défini par : $f(x_i) = i, f(y_j) = n + j$. En utilisant cette relation d'ordre $B_{[\vee(\wedge)]}(\Phi)$ correspond exactement à $\{(x_i \vee [y_1 \wedge y_2 \wedge \cdots \wedge y_m])|1 \leq i \leq n\}$. Evidemment, la base de transactions $\mathcal{D}^b_\Phi$ contient un unique itemset fréquent et fermé $\{y_1, y_2, \ldots, y_m\}$. L'application de notre algorithme conduit à la représentation compacte suivante $\Phi' = [\bigwedge_{1 \leq i \leq n}(x_i \vee z)] \wedge [\bigwedge_{1 \leq j \leq m}(\neg z \vee y_j)]$. On obtient exactement le même gain que dans [10] ($\mathcal{O}(n+m)$ clauses binaires et une variable supplémentaire).

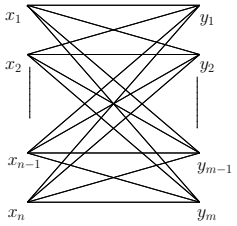


FIGURE 1 – Bi-clique : Représentation avec $n \times m$ clauses

Cliques de Clauses Binaires Soit $\Phi = \bigwedge_{1 \leq i \leq n-1}[(x_i \vee x_{i+1}) \wedge \cdots \wedge (x_i \vee x_n)]$ une clique de n^2 clauses binaires (voir Figure 2). L'ensemble $B_{[\vee(\wedge)]}(\Phi) = \{[x_i \vee (x_{i+1} \wedge \cdots \wedge x_n)]|1 \leq i \leq n-1\}$ en utilisant l'ordre défini par : $f(x_i) = i$. Si nous regardons de plus près D^b_Ψ, l'itemset fréquent et fermé I avec la plus grande valeur $\alpha(I)$ corresponds à $\{x_{n/2}, \ldots, x_n\}$.

Dans les premières $\frac{n}{2}$ lignes de D^b_Φ, I est remplacé par une nouvelle variable x et un nouvel ensemble de clauses binaires $[x \vee (x_{\frac{n}{2}} \wedge, \cdots \wedge x_n)])$ est ajouté, conduisant à deux sous-problèmes de taille $\frac{n}{2} + 1$. De toute évidence, le même traitement est effectué sur l'ensemble $B_{[\vee(\wedge)]}(\Phi)$. Par conséquent, le nombre de variables est définie par l'équation de récurrence suivante :

$$\mathcal{V}(n) = 2 \times \mathcal{V}(\frac{n}{2} + 1) + 1 \tag{1}$$

$$\mathcal{V}(6) = 1. \tag{2}$$

Le cas de base est atteint pour $n = 6$, où la dernière variable additionnelle est introduit pour représenter la conjonction $x_4 \wedge x_5 \wedge x_6$. Pour $n < 6$ aucune nouvelle variable n'est introduite parce qu'aucun itemset fréquent et fermé ne peut conduire à une réduction de la taille de la formule. Par conséquent, à partir de la solution de l'équation de récurrence précédente, nous obtenons que notre encodage introduit $\mathcal{O}(n)$ variables auxiliaires. En utilisant le même raisonnement, on obtient aussi la même complexité $\mathcal{O}(n)$ pour le nombre de clauses binaires. Cela correspond à la complexité obtenue dans [10].

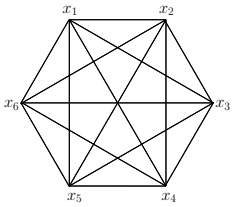


FIGURE 2 – Clique : Représentation avec n^2 clauses

Les deux cas particuliers de cliques et bi-cliques de clauses binaires considérées dans cette section permettent de montrer que lorsqu'une contrainte n'est pas bien codé, notre approche peut être utilisée pour corriger et produire un encodage plus efficace et compact de manière automatique.

6 Experimentations

Instance	orig.	comp.	% red
1dlx_c_iq57_a	190 Mb	164 Mb	13.68 %
6pipe_6_ooo.*-as.sat03-413	11 Mb	7.7 Mb	30.00 %
9dlx_vliw_at_b_iq6.*-*04-347	76 Mb	65 Mb	14.47 %
abb313GPIA-9-c.*.sat04-317	21 Mb	6.9 Mb	67.14 %
E05F18	3.7 Mb	2.2 Mb	40.54 %
eq.atree.braun.11.unsat	120 Kb	72 Kb	40.00 %
eq.atree.braun.12.unsat	144 Kb	88 Kb	38.88 %
k2mul.miter.*-as.sat03-355	1.5 Mb	1.3 Mb	13.33 %
korf-15	1.2 Mb	752 Kb	37.33 %
rbcl_xits_08_UNSAT	1.1 Mb	856 Kb	22.18 %
SAT_dat.k45	3.5 Mb	2.6 Mb	25.71 %
traffic_b_unsat	18 Mb	12 Mb	33.33 %
x1mul.miter.*-as.sat03-359	1.1 Mb	928 Kb	15.63 %
9dlx_vliw_at_b_iq3	19 Mb	15 Mb	21.05 %
9dlx_vliw_at_b_iq4	31 Mb	26 Mb	16.12 %
AProVE07-09	2.8 Mb	2.7 Mb	3.57 %
eq.atree.braun.10.unsat	96 Kb	56 Kb	41.66 %
goldb-heqc-frg1mul	348 Kb	328 Kb	5.74 %
minand128	7.7 Mb	2.6 Mb	66.23 %
ndhf_xits_09_UNSAT	2.6 Mb	2.1 Mb	19.23 %
velev-pipe-o-uns-1.1-6	5.5 Mb	4.4 Mb	20.00 %

TABLE 1 – Résultats de Mining4SAT : l'approche générale

Dans cette section, nous présentons une évaluation expérimentale de nos approches proposées. Deux types d'expérimentations ont été menées. La première porte sur la réduction de la taille de formules CNF arbitraires en utilisant l'algorithme Mining4SAT, tandis que la seconde tente de réduire la taille des sous-formules 2 -CNF uniquement, en utilisant l'algorithme Mining4Binary.

Les deux algorithmes sont testés sur différentes instances issues de la compétition SAT 2012. À partir des 600 instances issues de la catégorie applications, nous avons sélectionné 100 instances en prenant au moins une par famille. Tous les tests ont été effectués sur un cluster 3.2GHz Xeon (2 Go de RAM) et le temps limite a été fixé à 4 heures. Dans le tableau 1 et le tableau 2, nous indiquons la taille en Kilo-octets (`Kb`) ou méga-octets (`Mb`) de chaque instance SAT avant (`orig.`) et après réduction (`comp.`). Nous fournissons également le pourcentage de réduction $\%red$.

Le tableau 1 met en évidence les résultats obtenus par l'approche générale Mining4SAT. Dans ces expérimentations et afin de garantir des réductions possibles, nous nous limitons à chercher des itemsets fréquents et fermés de tailles supérieures ou égales à 4. Par conséquent, les clauses binaires ne sont pas considérés. Comme nous pouvons le remarquer, notre approche de réduction Mining4SAT permet des réductions en taille de plus de 20 % sur la majorité des cas. Notons aussi que le maximum (67,14 %) est atteint dans le cas de l'instance *abb313GPIA -9-c*sat04-317** : sa taille d'origine est de `21 Mb` et sa taille après réduction est de `6.9 Mb`.

Dans le tableau 2, nous présentons un échantillon des résultats obtenus par l'algorithme Mining4Binary en compactant uniquement les clauses binaires. Nous observons le même comportement que dans la première expérimentation en terme de réduction de la taille.

Pour étudier l'influence de la compression sur le temps de résolution, nous avons également testé le solveur SAT MiniSAT 2.2 à la fois sur les instances originales et sur celles obtenues après réduction.

En résumé, notre approche permet d'atteindre des réductions significatives en matière de taille des instances sans perdre en efficacité de résolution.

Instance	orig.	comp.	% red
velev-pipe-o-uns-1.1-6	5.5 Mb	3.2 Mb	41.81 %
9dlx_vliw_at_b_iq2	11 Mb	6 Mb	44.45 %
1dlx_c_iq57_a	190 Mb	124 Mb	34.73 %
7pipe_k	14 Mb	5.4 Mb	61.42 %
SAT_dat.k100.debugged	16 Mb	13 Mb	18.75 %
IBM_FV_2004_rule_batch _2_31_1_SAT_dat.k80.debugged	9.7 Mb	7.5 Mb	22.68 %
sokoban-sequential-p145-*.040-*	24 Mb	14 Mb	41.66 %
openstacks-*-p30_1.085-*	30 Mb	26 Mb	13.33 %
aaai10-planning-ipc5-*-12-step16	17 Mb	12 Mb	29.41 %
k2fix_gr_rcs_w8.shuffled	3.4 Mb	1.7 Mb	50.00 %
homer17.shuffled	20 Kb	16 Kb	20.00 %
gripper13u.shuffled-as.sat03-395	524 Kb	364 Kb	30.35 %
grid-strips-grid-y-3.045-*	52 Mb	42 Mb	19.23 %

TABLE 2 – Resultats de Mining4Binary : une approche pour 2-CNF

Dans nos expérimentations, nous avons présenté les deux algorithmes de compression, Mining4SAT (pour les clauses de taille arbitraire) et Mining4Binary (pour les formules 2-CNF). Comme les deux algorithmes peuvent être appliqués de manière indépendante, nous n'avons pas présenté les résultats en utilisant un algorithme en deux étapes de compression : application de l'algorithme de Mining4SAT générale dans la première étape sur la formule d'origine, suivi par l'al-

gorithme de Mining4Binary spécialisé sur la formule compressée dérivée dans le première étape. Les résultats peuvent être obtenus en cumulant les réductions obtenues dans les deux étapes. Par exemple, si nous prenons l' exemple de l'instance `1dlx_c_iq57_a` présente à la fois dans le tableau 1 et le tableau 2, la réduction de l'algorithme en deux étapes est calculée comme suit : $(190 - 164) + (190 - 124) = 92Mb$. Le rapport de réduction est égal à $48,42\%$.

7 Conclusion et travaux futurs

Dans cet article, nous proposons la première approche de fouille de données `Mining4SAT`, pour réduire de la taille des formules booléennes sous forme normale conjonctive (CNF). Elle peut être considérée comme une étape de pré-traitement qui a pour but de découvrir des connaissances structurelles cachées qui sont utilisées pour diminuer le nombre de littéraux et de clauses. Mining4SAT combine les deux techniques d'extraction de motifs fréquents pour découvrir des structures intéressantes.

Références

[1] R. Agrawal, T. Imielinski, and A. N. Swami. Mining association rules between sets of items in large databases. In *ACM SIGMOD International Conference on Management of Data*, pages 207–216, Baltimore, 1993. ACM Press.

[2] Emmanuel Coquery, Saïd Jabbour, Lakhdar Saïs, and Yakoub Salhi. A sat-based approach for discovering frequent, closed and maximal patterns in a sequence. In *Proceedings of the 20th European Conference on Artificial Intelligence (ECAI 2012)*, pages 258–263, 2012.

[3] Heidi E. Dixon, Matthew L. Ginsberg, David K. Hofer, Eugene M. Luks, and Andrew J. Parkes. Implementing a generalized version of resolution. In *Proceedings of the Nineteenth National Conference on Artificial Intelligence (AAAI 2004)*, pages 55–60, 2004.

[4] Mathieu L. Ginsberg and A. J. Parkes. Search, subsearch and qprop. In *Proceedings of the Seventh International Conference on Principles of Knowledge Representation and Reasoning (KR 2000)*, 2000.

[5] É. Grégoire, R. Ostrowski, B. Mazure, and L. Saïs. Automatic extraction of functional dependencies. In *Proceedings of the Seventh International Conference on Theory and Applications of Satisfiability Testing (SAT'04)*, pages 122–132, 2004.

[6] Dimitrios Gunopulos, Roni Khardon, Heikki Mannila, Sanjeev Saluja, Hannu Toivonen, and Ram Sewak Sharma. Discovering all most specific sentences. *ACM Trans. Database Syst.*, 28(2) :140–174, June 2003.

[7] Tias Guns, Siegfried Nijssen, and Luc De Raedt. Itemset mining : A constraint programming perspective. *Artif. Intell.*, 175(12-13) :1951–1983, 2011.

[8] David A. Plaisted and Steven Greenbaum. A structure-preserving clause form translation. *Journal of Symbolic Computation*, 2(3) :293–304, 1986.

[9] L. De Raedt, T. Guns, and S. Nijssen. Constraint programming for itemset mining. In *ACM SIGKDD*, pages 204–212, 2008.

[10] Jussi Rintanen. Compact representation of sets of binary constraints. In *Proceedings of the 17th European Conference on Artificial Intelligence (ECAI 2006)*, pages 143–147. IOS Press, 2006.

[11] A. Tiwari, RK Gupta, and DP Agrawal. A survey on frequent pattern mining : Current status and challenging issues. *Inform. Technol. J*, 9 :1278–1293, 2010.

[12] G.S. Tseitin. On the complexity of derivations in the propositional calculus. In H.A.O. Slesenko, editor, *Structures in Constructives Mathematics and Mathematical Logic, Part II*, pages 115–125, 1968.

[13] Takeaki Uno, Masashi Kiyomi, and Hiroki Arimura. Lcm ver. 2 : Efficient mining algorithms for frequent/closed/maximal itemsets. In Roberto J. Bayardo Jr., Bart Goethals, and Mohammed Javeed Zaki, editors, *FIMI*, volume 126 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2004.

[14] Guizhen Yang. The complexity of mining maximal frequent itemsets and maximal frequent patterns. In *Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '04, pages 344–353, New York, NY, USA, 2004.

[15] International sat competition. http ://www.satcompetition.org. Organized in conjunction with the International Conference on Theory and Applications of Satisfiability Testing.

Les contraintes table fragmentées : Combiner la compression et la réduction tabulaire

Nebras Gharbi **Fred Hemery** **Christophe Lecoutre** **Olivier Roussel**

CRIL - CNRS UMR 8188,
Université d'artois,
rue de l'université, 62307 Lens cedex, France

{gharbi,hemery,lecoutre,roussel}@cril.fr

Abstract

Many industrial applications require the use of table constraints (e.g., in configuration problems), sometimes of significant size. During the recent years, researchers have focused on reducing space and time complexities of this type of constraint. Static and dynamic reduction based approaches have been proposed giving new compact representations of table constraints and effective filtering algorithms. In this paper, we study the possibility of combining both static and dynamic reduction techniques by proposing a new compressed form of table constraints based on frequent pattern detection, and exploiting it in STR (Simple Tabular Reduction).

Résumé

De nombreuses applications industrielles nécessitent l'utilisation des contraintes table (les problèmes de configuration par exemple) ayant parfois une taille importante. Au cours des dernières années, les chercheurs ont mis l'accent sur la réduction de la complexité en espace et en temps de ce type de contraintes. Des approches basées sur la réduction statique et dynamique ont été proposées donnant de nouvelles représentations compactes des contraintes table et des algorithmes de filtrage efficaces. Dans cet article, nous étudions la possibilité de combiner à la fois des techniques de réduction statiques et dynamiques en proposant une nouvelle forme réduite des contraintes table basée sur la détection des motifs fréquents, et son exploitation dans STR (Simple Tabular Reduction).

Introduction

Les contraintes table, qui sont des contraintes en extension énumérant les tuples de valeurs autorisées ou interdites pour un ensemble de variables, sont largement étudiées en programmation par contraintes (CP). En effet, elles sont présentes dans de nombreux champs d'applications du monde réel, dans des domaines tels que la conception, la configuration, les bases de données et la modélisation de préférences. De plus, dans certains cas, les contraintes table fournissent pour un utilisateur non-expert la seule manière naturelle d'exprimer ses contraintes. Jusqu'à présent, la recherche sur les contraintes table a principalement porté sur le développement d'algorithmes rapides pour assurer la cohérence d'arc généralisée (GAC), qui est une propriété correspondant à un niveau maximum de filtrage lorsque les contraintes sont traitées indépendamment. Les algorithmes GAC pour les contraintes table sont l'objet d'un intérêt ancien, datant de GAC4 [21] et GAC-Schema [2]. Les algorithmes classiques itèrent sur l'ensemble des tuples de différentes manières [2, 19, 18]. Un algorithme récent basé sur AC5 a été proposé dans [20], et a montré son efficacité sur les contraintes table de petite arité. Pour les contraintes table de grande arité, le maintien dynamique de l'ensemble des supports des contraintes table peut être assuré à travers les variantes de STR [23, 15, 16].

Les contraintes table sont importantes pour la modélisation de problèmes sous contraintes, mais elles sont limitées en pratique car l'espace mémoire nécessaire pour les représenter peut croître de façon exponentielle avec leur arité. Pour réduire la complexité de l'espace, les chercheurs ont mis l'accent sur différentes formes de compression. Les arbre de préfixes [6], les diagrammes de décision multi-valeurs (MDD) [3] et les automates finis déterministes (DFAE) [22]

sont des structures générales utilisées pour représenter les contraintes table de façon compacte afin de faciliter le processus de filtrage. L'utilisation du produit cartésien est un autre mécanisme classique pour représenter de manière compacte un grand ensemble de tuples. Par exemple, il a été appliqué avec succès pour le traitement des ensembles de solutions [10], l'élimination de symétries [5, 4], et l'apprentissage [13, 17]. Jusqu'à présent, cette forme de compression a été utilisée par deux algorithmes GAC : en revisitant le schéma général GAC [14] et en combinant les tuples compressés avec STR [24]. Le dernier travail montre comment les variantes STR2 et STR3 peuvent avantageusement bénéficier de la forme compressée des tuples lorsque le taux de compression est élevé.

Récemment, nous avons proposé une approche de compression originale basée sur des algorithmes de *data-mining* [7], où toutes les occurrences des motifs les plus fréquents dans une contrainte sont remplacées par leurs indices dans une table des motifs. L'utilisation des techniques de fouille de données pour compresser les contraintes table a également été étudiée dans [11], mais d'une manière très différente car des variables et des valeurs supplémentaires sont nécessaires, et les contraintes sont reformulées. Les mêmes auteurs ont également étudié la compression des instances SAT dans [12]. Dans [7], un motif a été défini comme une séquence de valeurs consécutives. ce qui nous a empêché de bénéficier des variantes de STR optimisés. Dans cet article, nous proposons de relâcher cette condition (séquence de valeurs consécutives), en considérant tout sous-tuple comme un motif fréquent possible, et en identifiant les plus fréquents au moyen de techniques de fouille de données. Par conséquent, chaque table peut être "découpée" en plusieurs fragments, où chaque fragment associe à un motif μ une sous-table contenant toutes les extensions de μ qui peuvent être trouvées dans la table de départ.

Nous proposons un algorithme pour manipuler les contraintes table fragmentées basé sur STR2, la variante optimisée de STR. Le papier est organisé comme suit. Après le rappel des définitions usuelles en section 1, nous présentons, dans la section 2, un processus de compression pour des contraintes table en détaillant l'algorithme utilisé pour obtenir la nouvelle forme de contraintes table fragmentées. Ensuite, nous décrivons, dans la section 3, un algorithme optimisé pour assurer GAC sur les contraintes table fragmentées. Enfin, nous concluons après avoir donné quelques résultats expérimentaux en section 4.

1 Préliminaires

Un *réseau de contraintes* (discret) (CN) N est un ensemble fini de n variables reliées par un ensemble fini de e contraintes. Chaque *variable* x a un *domaine* qui représente l'ensemble fini de valeurs qui peuvent être assignées à x. Le domaine *initial* d'une variable x est noté $dom^{init}(x)$ tandis que le domaine *courant* de x est noté $dom(x)$. Nous avons toujours $dom(x) \subseteq dom^{init}(x)$. Chaque *contrainte* c porte sur un ensemble ordonné de variables, appelé *portée (scope)* de c et noté $scp(c)$, et est sémantiquement défini par une *relation*, notée $rel(c)$, qui contient l'ensemble des tuples autorisés pour les variables impliquées dans c. Une *contrainte table* (positive) c est une contrainte telle que $rel(c)$ est définie explicitement en énumérant les tuples qui sont autorisés par c (voir exemple ci-dessous). L'*arité* d'une contrainte c est la taille de $scp(c)$. L'arité maximale du réseau sera notée r.

Example 1 *Soit c une contrainte table positive portant sur les variables x_1, x_2, x_3, x_4, x_5 telles que $dom(x_1) = dom(x_2) = dom(x_3) = dom(x_4) = dom(x_5) = \{a, b, c\}$. La table 1 liste les 7 tuples autorisés par la contrainte c.*

	x_1	x_2	x_3	x_4	x_5
τ_1	(c,	b,	c,	a,	c)
τ_2	(a,	a,	b,	c,	a)
τ_3	(a,	c,	b,	c,	a)
τ_4	(b,	a,	c,	b,	c)
τ_5	(b,	a,	a,	b,	b)
τ_6	(c,	c,	b,	c,	a)
τ_7	(a,	c,	a,	c,	a)

TABLE 1 – Une contrainte table c portant sur x_1, x_2, x_3, x_4, x_5

Soit $X = \{x_1, \ldots, x_r\}$ un ensemble ordonné de variables. Une *instanciation* I de X est un ensemble $\{(x_1, a_1), \ldots, (x_r, a_r)\}$ qui est également noté $\{x_1 = a_1, \ldots, x_r = a_r\}$ tel que $\forall i \in 1..r, a_i \in dom^{init}(x_i)$. X est notée $vars(I)$ et chaque a_i est notée $I[x_i]$. Une instanciation I est *valide ssi* $\forall (x, a) \in I, a \in dom(x)$. Un *$r$-tuple* τ sur X est une suite de valeurs $(a_1, \ldots, a_r)$ telle que $\forall i \in 1..r, a_i \in dom^{init}(x_i)$; la valeur a_i sera notée $\tau[x_i]$. Un tuple défini sur un ensemble X peut être vu comme une instanciation de X et inversement. De ce fait, un r-tuple τ sur $scp(c)$ est *valide ssi* l'instanciation sous-jacente est valide. Un r-tuple τ sur $scp(c)$ est un *support* sur la contrainte r-aire c *ssi* τ est un tuple valide qui est autorisé par c. Si τ est un support sur une contrainte c impliquant une variable x et tel que $\tau[x] = a$, on dit que τ est un *support pour*

(x, a) sur c. GAC est une cohérence de domaine définie comme suit :

Definition 1 *Une contrainte c satisfait la* cohérence d'arc généralisée *(GAC) ssi $\forall x \in scp(c), \forall a \in dom(x)$, il existe au moins un support pour (x, a) sur c. Un CN N est GAC ssi chaque contrainte de N est GAC.*

L'application de GAC implique la suppression de toutes les valeurs qui n'ont pas de support sur une contrainte. De nombreux algorithmes ont été mis au point pour établir GAC selon la nature des contraintes. STR [23] est l'un de ces algorithmes pour les contraintes table : il supprime les tuples invalides lors de la recherche de supports en utilisant une structure de données qui sépare les tuples valides des tuples invalides. Cette méthode de recherche des supports améliore le temps de recherche en évitant les tests redondants sur des tuples invalides qui ont déjà été détectés comme invalides lors des précédents application de GAC. STR2 [15], une optimisation de STR, évite certaines opérations de base concernant la validité de tuples et l'identification des supports, par l'introduction de deux ensembles importants appelés S^{sup} et S^{val} (décrits ultérieurement). Dans le meilleur des cas, STR2 est r fois plus rapide que STR. Nous introduisons maintenant les notions de motif et sous-table qui seront utilisées dans notre approche.

Definition 2 *Un **motif** μ d'une contrainte c est une instanciation I d'un sous ensemble de variables de c. On note $scp(\mu)$ sa portée, qui est $vars(I)$, $|\mu|$ sa longueur, qui est égale à $|scp(\mu)|$, et $nbOcc(\mu)$ son nombre d'occurrences dans $rel(c)$, qui est $|\{\tau \in rel(c) \mid \mu \subseteq \tau\}|$.*

Example 2 *Dans la table 1, μ_1=$\{x_1 = a, x_4 = c, x_5 = a\}$ et μ_2=$\{x_3 = c, x_5 = c\}$ sont des motifs de longueurs respectives 3 et 2, tels que $scp(\mu_1)$=$\{x_1, x_4, x_5\}$ et $scp(\mu_2)$=$\{x_3, x_5\}$.*

Definition 3 *Une **sous-table** T associée à un motif μ d'une contrainte c est la table obtenue en ne conservant que les tuples de c qui contiennent μ et en effaçant μ dans chacun de ces tuples.*

$$T = \{\tau \setminus \mu \mid \tau \in rel(c) \wedge \mu \subseteq \tau\}$$

La portée de T est $scp(T) = scp(c) - scp(\mu)$

Example 3 *La table 2 représente la sous-table associée au motif μ_1=$(x_1 = a, x_4 = c, x_5 = a)$ de c décrite par la table 1.*

Definition 4 *Un fragment d'une contrainte c est un couple (μ, T) tel que μ est un motif de la contrainte c et T est la sous-table associée au motif μ.*

x_2	x_3
a	b
c	b
c	a

TABLE 2 – La sous-table T_1 associée au motif μ_1

Comme l'ensemble des tuples représentés par un fragment (μ, T) représente en fait le produit cartésien de μ par T, nous allons également utiliser la notation $\mu \otimes T$ pour désigner un fragment d'une contrainte. Après le processus de fragmentation d'une contrainte, l'ensemble des tuples qui ne sont associés à aucun motif sont regroupés dans un *fragment par défaut* noté $(\emptyset, T)$

Example 4 *Le motif μ=$(x_1 = a, x_4 = c, x_5 = a)$ de la contrainte c, détecté en figure 1(a), apparaît dans les tuples τ_2, τ_3 et τ_7. Donc le fragment résultant est composé du motif μ et sa sous-table correspondante extraite de c, comme décrit en figure 1(b).*

	x_1	x_2	x_3	x_4	x_5
τ_1	(c,	b,	c,	a,	c)
τ_2	(**a**,	a,	b,	**c**,	**a**)
τ_3	(**a**,	c,	b,	**c**,	**a**)
τ_4	(b,	a,	c,	b,	c)
τ_5	(b,	a,	a,	b,	b)
τ_6	(c,	c,	b,	c,	a)
τ_7	(**a**,	c,	a,	**c**,	**a**)

(a) Une contrainte c

x_1	x_4	x_5
a	c	a

$\otimes$

x_2	x_3	
a	b	τ_2
c	b	τ_3
c	a	τ_7

(b) Un fragment (μ, T) de c

FIGURE 1 – Un exemple de fragment de contrainte

Le test de validité sur les tuples (classiques ou bien compressés) est une opération très importante dans les algorithmes de filtrage pour les contraintes table. Pour les contraintes table fragmentée, nous étendons la notion de validité à un fragment d'une contrainte.

Definition 5 *Un fragment (μ, T) est* valide *ssi il existe au moins un tuple du produit cartésien $\mu \otimes T$ qui soit valide. De ce fait, un fragment est valide ssi son motif est valide et sa sous-table correspondante contient au moins un sous-tuple valide.*

2 Méthode de compression

Plusieurs algorithmes de fouille de données, tels que Apriori [1]et FP -Growth [8], peuvent être utilisés pour identifier les motifs les plus fréquents. Dans le cadre de notre approche, nous n'avons pas besoin d'identifier chaque motif fréquent possible mais seulement ceux qui sont utiles pour la compression, et en particulier au plus un motif par tuple. La construction d'un *FP-Tree* (*Frequent-Pattern Tree*) qui est la première étape dans l'algorithme *FP-Growth* est particulièrement bien adapté à cet objectif car elle identifie chaque motif long et fréquent. Cette construction ne nécessite que trois parcours de la table de la contrainte.

Nous expliquons brièvement la construction d'un *FP-Tree* dans notre contexte de compression de table, en utilisant la contrainte donnée par le tableau 1. Les détails de la méthode générale peuvent être trouvés dans [8, 9]. L'algorithme prend comme paramètre *minSupport* qui est le nombre minimal d'occurrences d'un motif pour qu'il soit considéré comme fréquent. Dans notre exemple, nous allons utiliser *minSupport*= 2 pour identifier les motifs qui apparaîssent au moins deux fois.

Dans une première étape, nous collectons le nombre d'occurrences de chaque valeur. Par abus de langage, nous appellerons *fréquence* le nombre d'occurrences d'une valeur. Cette étape nécessite un parcours de la table. Le résultat sur notre exemple est donné par la figure 2(a). Ensuite, lors d'un deuxième parcours, nous trions l'ensemble des tuples par ordre décroissant de fréquence des valeurs. Le résultat est donné par la figure 2(b) où la fréquence d'une valeur est donnée entre parenthèses. Les valeurs qui ont une fréquence en dessous du seuil *minSupport* sont retirées du tuple (elles sont identifiés en caractères gras) parce qu'ils ne peuvent pas apparaître dans un motif fréquent. Une fois le tuple trié et éventuellement réduit, il est inséré dans le FP-Tree qui est essentiellement un arbre de préfixe où chaque branche représente la partie fréquente d'un tuple et chaque nœud contient le nombre de branches qui partagent ce nœud. Chaque arête liant un parent à son enfant est étiquetée avec une valeur. Le nœud racine n'est pas étiqueté. La figure 3(a) représente le FP-Tree obtenu sur notre exemple. Le premier tuple inséré dans l'arbre est le début de τ_1 qui est $(x_1 = c, x_3 = c, x_5 = c)$. Cela crée la branche la plus à gauche de l'arbre. Chaque nœud de cette branche a initialement une fréquence de 1. Le deuxième tuple inséré est $(x_4 = c, x_5 = a, x_1 = a, x_2 = a, x_3 = b)$ qui crée la troisième branche à gauche dans l'arborescence (chaque nœud ayant une fréquence de 1 à cette étape). Lorsque τ_3 est insérée, la nouvelle branche $(x_4 = c, x_5 = a, x_1 = a, x_2 = c, x_3 = b)$ partage

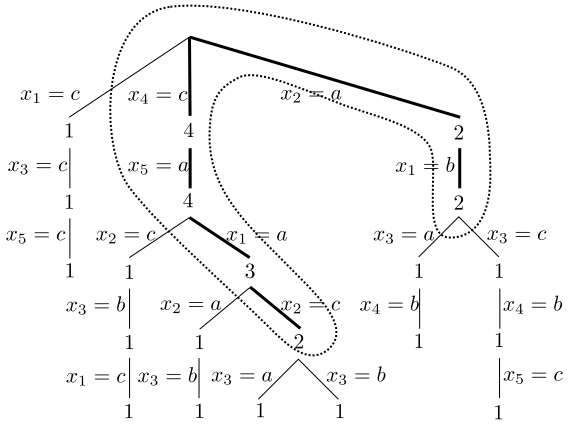
	x_1	x_2	x_3	x_4	x_5
a	3	3	2	1	4
b	2	1	3	2	1
c	2	3	2	4	2

(a) Les fréquences

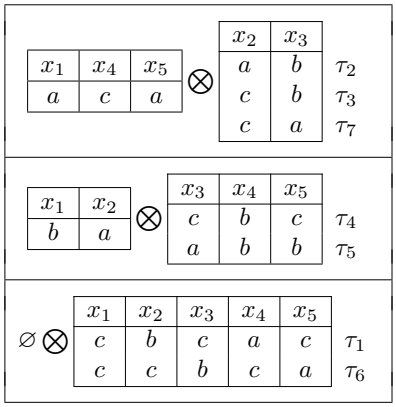
τ_1	(2) $x_1 = c$	(2) $x_3 = c$	(2) $x_5 = c$	(1) $\mathbf{x_2 = b}$	(1) $\mathbf{x_4 = a}$
τ_2	(4) $x_4 = c$	(4) $x_5 = a$	(3) $x_1 = a$	(3) $x_2 = a$	(3) $x_3 = b$
τ_3	(4) $x_4 = c$	(4) $x_5 = a$	(3) $x_1 = a$	(3) $x_2 = c$	(3) $x_3 = b$
τ_4	(3) $x_2 = a$	(2) $x_1 = b$	(2) $x_3 = c$	(2) $x_4 = b$	(2) $x_5 = c$
τ_5	(3) $x_2 = a$	(2) $x_1 = b$	(2) $x_3 = a$	(2) $x_4 = b$	(1) $\mathbf{x_5 = b}$
τ_6	(4) $x_4 = c$	(4) $x_5 = a$	(3) $x_2 = c$	(3) $x_3 = b$	(2) $x_1 = c$
τ_7	(4) $x_4 = c$	(4) $x_5 = a$	(3) $x_1 = a$	(3) $x_2 = c$	(2) $x_3 = a$

(b) Les tuples triées selon un ordre décroissant de fréquence

FIGURE 2 – Les deux première étapes de compression



(a) L'arbre des péfixes fréquents (FP-tree)



(b) Table compressée

FIGURE 3 – FP-tree et la table compressée

ses trois premières arêtes avec la dernière branche. De ce fait, les fréquences des nœuds correspondants sont incrémentées à 2. Les autres tuples sont insérés

de la même façon. Finalement, les nœuds ayant une fréquence en dessous du seuil *minSupport* sont supprimés. L'arbre restant est représenté avec des lignes épaisses et entouré par une ligne en pointillés sur la figure 3(a). Il faut maintenant identifier les motifs du FP-tree qui sont utiles pour la compression. Chaque nœud de l'arbre correspond à un motif fréquent μ qui peut être lu sur le chemin menant de la racine au nœud lui-même. La fréquence f de ce motif est donnée par le nœud lui-même. Les gains qui peuvent être obtenus par la factorisation de ce motif fréquent est $|\mu| \times (f-1)$ valeurs (on peut supprimer toutes les occurrences du motif, sauf une). Dans notre exemple, pour la première branche en gras, nous pouvons voir qu'en utilisant le motif $(x_4 = c, x_5 = a)$, nous gagnons six valeurs, avec le motif $(x_4 = c, x_5 = a, x_1 = a)$ nous gagnons également 6 valeurs tandis qu'on ne gagne que 4 valeurs avec la branche complète $(x_4 = c, x_5 = a, x_1 = a, x_2 = c)$. Par conséquent, nous réduisons l'arbre en supprimant les nœuds qui offrent un gain moindre que leur père. Les feuilles de l'arbre obtenu représentent les motifs fréquents utilisé dans notre compression : $(x_4 = c, x_5 = a, x_1 = a)$ et $(x_2 = a, x_1 = b)$. Pour terminer, nous créons un fragment pour chaque motif fréquent que nous avons identifié. Ces fragments seront remplis lors d'un dernier parcours de la contrainte. Pour chaque tuple, nous utilisons le FP-tree pour identifier si le tuple (trié) commence par un motif fréquent. Dans ce cas, nous ajoutons le reste du tuple à la *sous-table* correspondante. Les tuples qui ne commencent pas par un motif fréquent sont ajoutés au fragment par défaut.

L'algorithme 1 résume les différentes étapes du processus de compression.

Algorithm 1: compress(T: table, minSupport: entier)

```
1  calculer la fréquence de chaque valeur de T
2  for i ∈ 1..|T| do
3  |   τ ← T[i]
4  |   trier τ par ordre décroissant de fréquence et
   |   supprimer les valeurs moins fréquentes que le
   |   seuil minSupport
5  |   ajouter τ à l'arbre des préfixes fréquents (ceci
   |   met à jour la fréquence des nœuds)
6  |   tmp[i] ← τ
7  réduire l'arbre en supprimant les nœuds qui ont
   une fréquence en dessous de minSupport ou bien
   tels que |μ| × (f − 1) est plus petit que pour leur
   parent
8  for i ∈ 1..|T| do
9  |   τ ← tmp[i]
10 |   chercher dans l'arbre si τ commence par un
   |   motif fréquent μ. Si ce n'est pas le cas, μ ← ∅
11 |   ajouter T[i] \ μ à la sous-table correspondante
   |   à μ
```

3 Processus de filtrage des contraintes tables fragmentées

Afin d'appliquer GAC sur les contraintes table fragmentées, notre idée est d'adapter la technique (STR), et plus précisément la variante STR2 optimisée. Comme une contrainte table fragmentée est composée de plusieurs fragments, chacune composée d'un motif et d'une *sous-table*, le processus de filtrage que nous proposons agit à deux niveaux distincts. Au niveau haut, la validité de chaque fragment est déterminée, et au niveau bas, la validité de chaque couple (motif, sous-tuple) est vérifiée. Rappelons qu'un fragment est valide *ssi* à la fois son motif est valide et au moins un sous-tuple de la sous-table est valide (voir la définition 5). Dans cette section, nous décrivons d'abord les structures de données utilisées, puis nous présentons notre algorithme GAC, et enfin nous donnons une illustration.

3.1 Structures de données

Une contrainte table fragmentée c est représentée par un tableau `entries`$[c]$ de p fragments. La gestion des fragments valides, appelés fragments *courants*[1], est effectuée comme suit :

- `entriesLimit`$[c]$ est l'indice du dernier fragment courant dans `entries`$[c]$. Les éléments de `entries`$[c]$ aux indices allant de 1 à `entriesLimit`$[c]$ sont les fragments courants de c.
- la suppression d'un fragment (qui est devenu invalide) à l'indice i est faite par l'appel à la fonction removeEntry(c, i). Cet appel effectue une permutation entre les fragments d'indice i et `entriesLimit`$[c]$, puis décrémente `entriesLimit`$[c]$. Notez que l'ordre initial de fragments n'est pas conservé.
- La restauration d'un ensemble de fragments se fait simplement en changeant la valeur de `entriesLimit`$[c]$.

Chaque fragment de `entries` peut être représenté comme un enregistrement composé d'un champ `pattern` et un champ `subtable`. Plus précisément :

- le champ `pattern` enregistre une instanciation partielle μ, et peut être représenté en pratique comme une structure composée de deux tableaux : un pour les variables, la portée du motif, et l'autre

1. Les fragments courants correspondent aux fragments valides obtenus comme résultat du dernier appel de l'algorithme.

pour les valeurs.
- la structure `subtable` stocke une sous-table T et peut être représentée en pratique comme une structure composée de deux tableaux : un pour les variables, à savoir la portée de la sous-table T, et d'autre part, un tableau à deux dimensions, pour les sous-tuples.

Dans notre présentation, nous allons traiter directement μ et T sans tenir compte de tous les détails d'implantation. Par exemple, T sera considéré comme un tableau à deux dimensions. La gestion de l'ensemble des sous-tuples valides, appelés sous-tuples *courants* de T, est réalisée comme suit :

- $limit[T]$ est l'indice du dernier sous-tuple courant dans T. Les éléments de T aux indices allant de 1 à $\texttt{limit}[T]$ sont les sous-tuples courants de T.
- la suppression d'un sous-tuple (qui est devenu invalide) à l'indice i est effectuée par un appel à la fonction removeSubtuple(T, i). Cet appel effectue une permutation entre les sous-tuples à indices i et $\texttt{limit}[T]$, puis décrémente $\texttt{limit}[T]$. Notez que l'ordre initial de sous-tuples n'est pas conservé.
- la restauration d'un ensemble de sous-tuples se fait simplement en changeant la valeur de $\texttt{limit}[T]$.

Notez que la gestion des fragments courants et sous-tuples courants utilise les mêmes principes que ceux de STR. En outre, comme dans [15], nous utilisons deux ensembles de variables, appelés S^{val} et S^{sup}. L'ensemble S^{val} contient des variables non affectées (et éventuellement, la dernière variable assignée) dont les domaines ont été réduits depuis le dernier appel de l'algorithme de filtrage sur c. Pour configurer S^{val}, nous avons besoin d'enregistrer la taille du domaine de chaque variable x juste après l'exécution de STR-slice sur c : cette valeur est enregistrée dans $\texttt{lastSize}[x]$. L'ensemble S^{sup} contient des variables non affectées de la contrainte c dont les domaines contiennent chacun au moins une valeur pour laquelle un support doit être trouvé. Ces deux ensembles permettent de restreindre les itérations sur les variables à celles qui sont uniquement pertinentes. Nous utilisons également un tableau $\texttt{gacValues}[x]$ pour chaque variable x. A tout moment, $\texttt{gacValues}[x]$ contient toutes les valeurs de $dom(x)$ pour lesquelles un support a déjà été trouvé : par conséquent, les valeurs d'une variable x pour lesquelles on n'a pas trouvé de support sont exactement ceux de $dom(x) \setminus \texttt{gacValues}[x]$. Notez que les ensembles S^{val} and S^{sup} sont initialement définis par rapport à la portée de c. Cependant, pour chaque sous-table, nous aussi allons utiliser des ensembles locaux S^{lval} et S^{lsup} de S^{val} et S^{sup} comme expliqué plus loin.

3.2 Algorithmes

L'algorithme 2 est une procédure de filtrage, appelé STR-slice, qui établit GAC sur une contrainte table fragmentée c appartenant à un CN N. Les lignes 1–10, qui sont exactement les mêmes que ceux de l'algorithme 5 de [15], nous permettent d'initialiser les ensembles S^{val}, S^{sup} et `gacValues`. Rappelons que S^{val} doit contenir la variable courante, notée lastPast(P), si elle appartient à la portée de c. Les lignes 11–22 itèrent sur tous les fragments de c. Pour tester la validité d'un fragment, nous vérifions d'abord la validité du motif μ (algorithme 3), puis si le motif est valide, nous vérifions la validité de la sous-table T en la parcourant (algorithme 4). Si un fragment n'est plus valide, il est retiré à la ligne 22. Sinon, compte tenu des valeurs qui sont présentes dans le motif, nous devons mettre à jour `gacValues` ainsi que S^{sup} quand un premier support pour une variable est trouvé. Les lignes 23–30, qui sont exactement les mêmes que ceux de l'algorithme 5 de [15], permettent de gérer la réduction de domaines : les valeurs qui n'ont pas de support sont supprimées à la ligne 25 et si le domaine d'une variable x est vide, une exception est levée à la ligne 27. En outre, l'ensemble des variables X_{evt} réduits par STR-slice est calculé et retourné afin que ces "événements" puissent être propagés à d'autres contraintes.

L'algorithme 4 est une fonction importante, appelée scanSubtable, de STR-slice. Son rôle est de parcourir tous les sous-tuples courants d'une sous-table donnée, afin de recueillir des valeurs qui ont des supports et de supprimer les tuples non valides. Notez que lorsque cette fonction est appelée, nous avons la garantie que le motif associé à la sous-table est valide (notez l'opérateur "and then" à la ligne 14 de l'algorithme 2). La première partie de la fonction, plus précisément les lignes 1–10, nous permettent de construire des ensembles locaux S^{lval} et S^{lsup} de S^{val} et S^{sup}. Ces ensembles sont obtenus, respectivement, par l'intersection de S^{val} avec $scp(T)$ et S^{sup} avec $scp(T)$. Une fois les ensembles S^{lval} et S^{lsup} initialisés, nous bénéficions d'une exploitation optimale concernant les tests de validité et la recherche des supports, comme dans STR2. La deuxième partie de la fonction, les lignes 9–21, consiste à itérer sur l'ensemble des sous-tuples courants de T. Il s'agit d'un parcours d'un ensemble de tuples similaire au parcours classique utilisé dans STR2. Enfin, la ligne 22 renvoie vraie quand il existe encore au moins un sous-tuple valide. Il est intéressant de noter la synchronisation paresseuse effectuée entre l'ensemble global unique S^{sup} et les ensembles locaux spécifiques S^{lsup} (un ensemble par sous-table). Quand une variable x est identifiée comme "entièrement supportée" (pour toute valeur du domaine courant de x et pour toute contrainte c impliquant x, il existe un sup-

Algorithm 2: STR-slice(c: contrainte)

Entrées: c est une contrainte table fragmentée
Sorties: l'ensemble des variables dans $scp(c)$ avec des domaines réduits

```
   // Initialisation de S^val et S^sup
1  S^val ← ∅
2  S^sup ← ∅
3  if lastPast(P) ∈ scp(c) then
4      S^val ← S^val ∪ {lastPast(P)}
5  foreach variable x ∈ scp(c) | x ∉ past(P) do
6      gacValues[x] ← ∅
7      S^sup ← S^sup ∪ {x}
8      if |dom(x)| ≠ lastSize[c][x] then
9          S^val ← S^val ∪ {x}
10         lastSize[c][x] ← |dom(x)|
   // Itération sur l'ensemble des fragments
11 i ← 1
12 while i ≤ entriesLimit[c] do
13     (μ, T) ← entries[c][i]    // ith fragment
14     if isValidPattern(μ) and then
       scanSubtable(T) then
15         foreach variable x ∈ scp(μ) | x ∈ S^sup do
16             if μ[x] ∉ gacValues[x] then
17                 gacValues[x] ←
                   gacValues[x] ∪ {μ[x]}
18                 if |dom(x)| = |gacValues[x]| then
19                     S^sup ← S^sup \ {x}
20         i ← i + 1
21     else
22         removeEntry(c, i)      // entriesLimit[c]
           est décrementé
   // Les domaines sont mis à jour et X_evt
      est calculé, comme dans STR2
23 X_evt ← ∅
24 foreach variable x ∈ S^sup do
25     dom(x) ← gacValues[x]
26     if dom(x) = ∅ then
27         throw INCONSISTENCY
28     X_evt ← X_evt ∪ {x}
29     lastSize[c][x] ← |dom(x)|
30 return X_evt
```

Algorithm 3: isValidPattern(μ: motif): booléen

```
1 foreach variable x ∈ scp(μ) do
2     if μ[x] ∉ dom(x) then
3         return false
4 return true
```

Algorithm 4: scanSubtable(T: sous-table): booléen

Entrées: T est une sous-table d'un fragment
Sorties: vrai ssi il existe au moins un tuple valide dans la sous-table T

```
   // Initialisation de S^lval et S^lsup
1  S^lval ← ∅
2  foreach variable x ∈ S^val do
3      if x ∈ scp(T) then
4          S^lval ← S^lval ∪ {x}
5  S^lsup ← ∅
6  foreach variable x ∈ S^sup do
7      if x ∈ scp(T) then
8          S^lsup ← S^lsup ∪ {x}
   // Itération sur l'ensemble des
      sous-tuples
9  i ← 1
10 while i ≤ limit[T] do
11     τ ← T[i]    // i^ème sous-tuple courant
12     if isValidSubtuple(S^lval, τ) then
13         foreach variable x ∈ S^lsup do
14             if τ[x] ∉ gacValues[x] then
15                 gacValues[x] ←
                   gacValues[x] ∪ {τ[x]}
16                 if |dom(x)| = |gacValues[x]| then
17                     S^lsup ← S^lsup \ {x}
18                     S^sup ← S^sup \ {x}
19         i ← i + 1
20     else
21         removeSubtuple(T, i))    // limit[T] est
           décrémenté
22 return limit[T] > 0
```

port pour x dans c), elle est immédiatement retirée de S^{sup} (voir la ligne 19 de l'algorithme 2 et la ligne 18 de l'algorithme 4). Par conséquent, cela signifie que les prochaines sous-tables (fragments) bénéficieront d'une telle réduction, mais l'information est transmise uniquement à l'initialisation (lignes 6–8 de l'algorithme 4). D'autre part, une fois initialisé, l'ensemble global S^{val} n'est jamais modifié lors de l'exécution de *STR-slice*.

Comme GAC-slice est une extension directe de STR2, il maintient GAC.

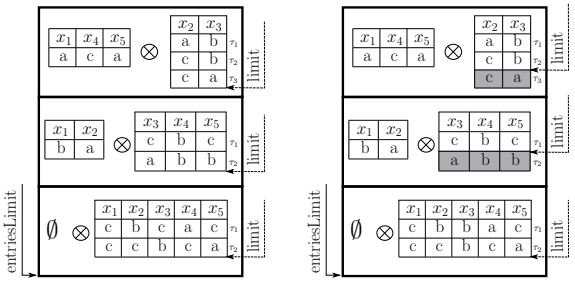
Au sujet du retour arrière : Dans notre implantation, les fragments et les tuples peuvent être restaurés en modifiant la valeur de la limite des pointeurs (`entriesLimit`$[c]$ et `limit`$[T]$ pour chaque sous-table

Algorithm 5: isValidSubtuple(S^{lval}: variables, τ: tuple): booléen

```
1 foreach variable x ∈ S^lval do
2     if τ[x] ∉ dom(x) then
3         return false
4 return true
```

T de c), enregistrée à chaque profondeur de la recherche. La restauration est alors réalisée en $O(1+p)$ (pour chaque contrainte) où p est le nombre de fragments. Cependant, en introduisant une structure de données simple, il est possible de n'appeler la procédure de restauration qu'en cas de besoin, ce qui limite la complexité de la restauration à $O(1)$ dans certains cas : il suffit d'enregistrer les pointeurs de limites qui doivent être mis à jour lors d'un retour en arrière, et ce pour chaque niveau. Lorsque l'algorithme de recherche procède à un retour en arrière, nous devons également gérer l'ensemble `lastSize`. Comme indiqué dans [15], nous pouvons enregistrer le contenu d'un tel réseau à chaque profondeur de la recherche, de sorte que l'état d'origine de la table peut être restauré lors d'un retour en arrière.

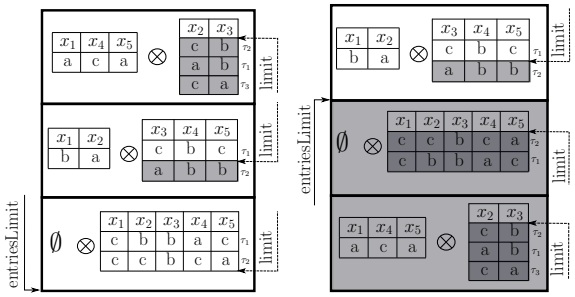
3.3 Illustration



(a) Avant l'appel (b) Après l'appel

FIGURE 4 – L'appel de STR-slice sur une contrainte table fragmentée après l'événement $x_3 \neq a$

Les figures 4 et 5 illustrent les différentes étapes pour filtrer une contrainte table fragmentée, quand STR-slice est appelé après un événement. Dans la figure 4, considérant que le nouvel événement est tout simplement $x_3 \neq a$ (soit la suppression de la valeur a de $dom(x_3)$), STR-slice commence par vérifier la validité des fragments courants (de 1 à `entriesLimit`). Ainsi, pour le premier fragment, la validité du motif μ=$\{x_1 = a, x_4 = c, x_5 = a\}$ est vérifiée en premier lieu. Puisque μ reste encore valide (notre hypothèse est que l'événement était seulement $x_3 \neq a$), la sous-table du premier fragment est parcourue. Dans ce cas, le sous-tuple $\{x_2 = c, x_3 = a\}$ est déclaré invalide, ce qui modifie la valeur de `limit` pour la sous-table de ce premier fragment. Après l'appel de STR-slice, la contrainte est dans l'état décrit par la figure 4(b). Dans la figure 5, le nouvel événement est $x_3 \neq b$. On recommence avec le premier fragment courant et puisque le motif est toujours valable, nous vérifions la validité des sous-tuples associés. Comme le sous- tuple $\{x_2 = a, x_3 = b\}$ n'est plus valide, il est permuté avec $\{x_2 = c, x_3 = b\}$. Ce sous-tuple est à son tour détecté invalide, ce qui fait passer la valeur de `limit` à 0. Ceci est illustré en figure 5(a). Comme la sous-table du premier fragment est vide, le fragment est supprimé en permutant sa position avec celle du dernier fragment courant. Après l'appel de STR-slice, l'état de la contrainte est telle que donné par la figure 5(b) (notez qu'une deuxième permutation a été effectuée)



(a) Après le parcours du premier fragment (b) Après l'appel

FIGURE 5 – à partir de la figure 4(b), STR-slice est appelé après l'événement $x_3 \neq b$

4 Résultats expérimentaux

Afin de montrer l'intérêt pratique de notre approche, nous avons mené une expérimentation avec notre solveur AbsCon en utilisant un cluster avec des bi-processeurs Xeon à quatre cœurs à 2.66 GHz avec 16GiB de RAM sous Linux. Comme STR1, STR2 et STR3 appartiennent à l'état de l'art des algorithmes de GAC pour les contraintes table, nous comparons les comportements respectifs de STR1, STR2, STR3 et STR-slice sur différentes instances [2] impliquant des contraintes table positives d'arité supérieure à 2. Pour STR-slice, nous sélectionnons les motifs fréquents avec une fréquence au moins égale à 10 % du nombre de tuples dans la table. Cette valeur a été obtenue après

2. disponibles sur `http://www.cril.univ-artois.fr/CSC09`.

plusieurs expérimentations. D'une manière similaire on a choisi 10 comme taille minimale des sous-tables. Le paramétrage automatique de ces paramètres est l'une de nos perspectives. Nous utilisons MAC avec *dom/ddeg* comme choix d'ordonnancement des variables et *lexico* comme heuristique de choix de valeur, pour résoudre tous ces problèmes. Un temps maximal de 1200 secondes a été fixé par instance. Les deux heuristiques de choix garantissent l'exploration du même arbre de recherche, indépendamment de l'algorithme de filtrage utilisé.

Instance	#ins	STR1	STR2	STR3	STR-slice
a7-v24-d5-ps05	11	298.05	147.73	189.14	**115.30** (66% – 5.74)
bdd	70	44.53	**13.44**	99.21	20.35 (86% – 0.59)
crossword-ogd	43	90.05	39.35	**25.69**	29.59 (75.51% – 0.36)
crossword-uk	43	95.20	45.88	**44.33**	47.21 (88.69% – 0.18)
renault	46	19.66	14.39	**13.37**	17.20 (47.15% – 0.67)

TABLE 3 – Le temps CPU moyen sur différentes séries

Instance	STR1	STR2	STR3	MDD	STR-slice
a7-v24-d5-ps0.5-psh0.7-9	879	334	367	**25.5**	200 (69% – 5.41)
a7-v24-d5-ps0.5-psh0.9-6	353	195	324	**16.6**	174 (62% – 5.82)
bdd-21-2713-15-79-11	78.5	**23.5**	48.5	82.6	31.7 (88.05% – 0.28)
crossword-ogd-vg12-13	799	342	**208**	$> 1,200$	242 (73.46% – 0.74)
crossword-uk-vg10-13	1,173	**576**	589	$> 1,200$	598 (89.63% – 0.48)

TABLE 4 – Le temps CPU sur quelques instances

Le tableau 3 montre les résultats en moyenne (temps CPU en secondes) par série. Pour chaque série, le nombre d'instances testées est donné par #ins ; il correspond au nombre d'instances résolues par les trois variantes en 1200 secondes. Notez que la moyenne des taux de compression et le temps CPU (en secondes) sont également donnés pour STR-slice entre parenthèses. On définit le taux de compression comme la taille de la table fragmentée sur la taille initiale de la table, où la taille d'une table fragmentée est le nombre de valeurs sur l'ensemble des motifs et sous-tables. Les résultats du tableau 3 montrent que STR-slice est compétitif avec STR2 et STR3. Bien que le taux de compression obtenu pour les instances de la série *renault* soit plutôt encourageant, le temps CPU obtenu pour STR-slice est décevant. Nous pensons que la présence de nombreuses contraintes avec des petites tables dans les instances *renault* est pénalisant pour STR-slice parce que, dans ce cas, la gestion des fragments de la contrainte n'est pas compensée par la légère réduction spatiale obtenue. Le tableau 4 présente les résultats obtenus sur certaines instances. En termes d'espace, STR3 est la variante qui utilise la plus grande quantité de mémoire (parfois avec un facteur très important). STR-slice, bien que nécessitant quelques structures de données supplémentaires est la variante la moins chère de STR en terme de mémoire (comme indiqué par les taux de compression dans les tableaux 3 et 4). Notez que d'autres méthodes de compression de la littérature telles que celles basées sur MDD [3] peuvent surclasser les variantes STR lorsque le taux de compression est (très) élevé. C'est le cas, par exemple, sur les instances de la série *a7-v24-d5-ps05*. Cependant, sur d'autres séries telles que *crossword*, l'approche MDD peut être dépassée par un facteur important par les variantes de STR (sur les instances de mots croisés difficiles, STR2, STR3 et STR-slice sont généralement environ 5 fois plus rapide que MDD).

Une observation générale de cette expérimentation préliminaire est que STR-slice concurrence STR2 et STR3, sans toutefois l'emporter clairement. Plusieurs perspectives pour améliorer ce constat sont développées dans la conclusion.

Conclusion

Dans cet article, nous avons présenté une approche originale pour le filtrage des contraintes table : elle combine une nouvelle technique de compression en utilisant le concept de contraintes table fragmentées et une adaptation optimale de la réduction tabulaire (comme dans STR2). Notre expérimentation préliminaire montre que STR-slice est un concurrent aux algorithmes de l'état de l'art de la technique STR. Plusieurs améliorations de l'algorithme STR-slice sont envisageables. Tout d'abord, nous pensons que le réglage des paramètres utilisés pour guider la compression doit être automatisé (éventuellement, en utilisant des techniques d'apprentissage automatique). STR-slice pourrait alors bénéficier d'une meilleure compression. Deuxièmement, nous croyons que, dans le contexte de l'émergence du big data, de nouveaux problèmes de contraintes devraient émerger rapidement où les contraintes pourraient être de (très) grande arité et impliquer de très grandes tables. STR-slice pourrait avantageusement traiter ces "énormes" contraintes, surtout si l'on considère que la fragmentation pourrait être menée de manière récursive sur les sous-tables (une autre perspective de ce travail). Enfin, nous pensons que le concept de contraintes table fragmentées est intéressant en soi pour la modélisation. En effet, certaines formes d'expressions conditionnelles peuvent être représentées d'une manière simple et naturelle, directement avec les contraintes table fragmentées.

Remerciements

Ce travail a été soutenu à la fois par le CNRS et OSEO dans le cadre du projet ISI "Pajero".

Références

[1] R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. In *Proceedings of the 20th International Conference on Very Large Data Bases*, pages 487–499, 1994.

[2] C. Bessiere and J. Régin. Arc consistency for general constraint networks: preliminary results. In *Proceedings of IJCAI'97*, pages 398–404, 1997.

[3] 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.

[4] T. Fahle, S. Schamberger, and M. Sellman. Symmetry breaking. In *Proceedings of CP'01*, pages 93–107, 2001.

[5] F. Focacci and M. Milano. Global cut framework for removing symmetries. In *Proceedings of CP'01*, pages 77–92, 2001.

[6] 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.

[7] N. Gharbi, F. Hemery, C. Lecoutre, and O. Roussel. Optimizing STR algorithms with tuple compression. In *Proceedings of JFPC'13*, pages 143–146, 2013.

[8] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *Proceedings of SIGMOD'00*, pages 1–12, 2000.

[9] J. Han, J. Pei, Y. Yin, and R. Mao. Mining frequent patterns without candidate generation: A frequent-pattern tree approach. *Data Mining and Knowledge Discovery*, 8(1):53–87, 2004.

[10] P.D. Hubbe and E.C. Freuder. An efficient cross product representation of the constraint satisfaction problem search space. In *Proceedings of AAAI'92*, pages 421–427, 1992.

[11] S. Jabbour, L. Sais, and Y. Salhi. A mining-based compression approach for constraint satisfaction problems. *CoRR*, abs/1305.3321, 2013.

[12] S. Jabbour, L. Sais, Y. Salhi, and T. Uno. Mining-based compression approach of propositional formulae. In *Proceedings of CIKM'2013*, pages 289–298, 2013.

[13] G. Katsirelos and F. Bacchus. Generalized nogoods in CSPs. In *Proceedings of AAAI'05*, pages 390–396, 2005.

[14] G. Katsirelos and T. Walsh. A compression algorithm for large arity extensional constraints. In *Proceedings of CP'07*, pages 379–393, 2007.

[15] C. Lecoutre. STR2: Optimized simple tabular reduction for table constraints. *Constraints*, 16(4):341–371, 2011.

[16] C. Lecoutre, C. Likitvivatanavong, and R. Yap. A path-optimal GAC algorithm for table constraints. In *Proceedings of ECAI'12*, pages 510–515, 2012.

[17] C. Lecoutre, L. Sais, S. Tabary, and V. Vidal. Transposition Tables for Constraint Satisfaction. In *Proceedings of AAAI'07*, pages 243–248, 2007.

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

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

[20] J.-B. Mairy, P. van Hentenryck, and Y. Deville. An optimal filtering algorithm for table constraints. In *Proceedings of CP'12*, pages 496–511, 2012.

[21] R. Mohr and G. Masini. Good old discrete relaxation. In *Proceedings of ECAI'88*, pages 651–656, 1988.

[22] G. Pesant. A regular language membership constraint for finite sequences of variables. In *Proceedings of CP'04*, pages 482–495, 2004.

[23] J.R. Ullmann. Partition search for non-binary constraint satisfaction. *Information Science*, 177:3639–3678, 2007.

[24] W. Xia and R. Yap. Optimizing STR algorithms with tuple compression. In *Proceedings of CP'13*, pages 724–732, 2013.

L'outil *coupure* pour les QCSP

Vincent Barichard and Igor Stéphan
LERIA, Université d'Angers, France
{barichard,stephan}@info.univ-angers.fr

Résumé

Nous introduisons un nouvel outil permettant d'élaguer des branches satisfiables d'un arbre de recherche d'un QCSP. Cet outil permet, entre autres, de restaurer la propriété d'absorption du vrai par rapport à la disjonction dans les solveurs QCSP basés sur une conjonction de contraintes et un algorithme de recherche quantifié. Ce nouvel outil est inspiré de la coupure de Prolog comme un outil dont l'utilisation est laissée sous la responsabilité du concepteur du QCSP pour élaguer des parties de l'espace de recherche qui sont connues par construction inutiles à parcourir. Ce nouvel outil permet en particulier d'utiliser efficacement un QCSP pour spécifier des jeux à deux joueurs sans restreindre le langage QCSP. Notre solveur QCSP construit au-dessus de GeCode obtient d'excellents résultats vis-à-vis de l'état de l'art des solveurs QCSP.

Abstract

We introduce a new tool to prune some satisfiable branches of the search tree of a QCSP. This tool can be used to restore the annihilator property of true for disjunction in QCSP solver based on a conjunction of constraints and a quantified search algorithm. This new tool is inspired by the cut rule of Prolog as a tool under responsibility of the designer of the QCSP to prune some parts of the search space which are by construction known to be useless. It is a simple solution to use efficiently QCSP to design finite two-player games with out restricting the QCSP language. Our QCSP solver built over GeCode obtained very good results compared to state-of-the-art QCSP solvers.

1 Introduction

Le problème de satisfaction de contraintes quantifiées (ou QCSP pour *Quantified Constraint Satisfaction Problem*) est une généralisation du problème de satisfaction de contraintes (ou CSP pour *Constraint Satisfaction Problem*) dans laquelle les variables peuvent être non seulement quantifiées existentiellement (comme dans les CSP) mais aussi universellement [2]. L'étude des QCSP est récente mais il existe un réel intérêt pour mettre au point des techniques efficaces pour résoudre de tels problèmes (cf. [6] pour un panorama). Cette extension est pleine de promesses car elle permet de coder de manière plus compacte certains problèmes et même d'en modéliser d'autres qui ne peuvent l'être en CSP. Mais cette extension accroît la complexité de décision de NP-complet à PSPACE-complet. Les QCSP sont bien adaptés à la représentation des jeux à deux joueurs. Le domaine des QCSP est à l'intersection de deux domaines : le domaine des CSP [7] et celui des QBF [3]. Il n'y a, à l'aune de nos connaissances que trois solveurs QCSP, excepté le nôtre : BlockSolve[8], Queso[4] et Qecode [1]. BlockSolve est basé sur un algorithme de Fourier-Motzkin par élimination de quantificateurs. Queso est basé sur un algorithme de recherche quantifié et son approche est très proche de la nôtre. Qecode est aussi un solveur basé sur un algorithme de recherche quantifié mais est dédié à une restriction des QCSP (QCSP+), qui nécessite une forme restreinte de la quantification, et qui est utilisé principalement pour spécifier des jeux finis à deux joueurs. Ce dernier solveur est toujours maintenu tandis que les deux autres ne le sont plus. Comme un solveur QCSP basé sur un algorithme de recherche quantifié peut être vu comme une extension d'un solveur CSP basé sur un algorithme de recherche, il est particulièrement pertinent de chercher à construire un solveur QCSP au-dessus de technologies pour solveur CSP sans les changer. Cette approche est la nôtre : nous implantons QuaCode, un solveur QCSP, au-dessus de GeCode, une bibliothèque de classes pour la gestion des contraintes dans un CSP, sans changer le cœur de la bibliothèque.

Pour un solveur QCSP conçu comme une extension à un solveur CSP, l'ensemble des contraintes est considéré comme une conjonction (un $\wedge$-solveur QCSP). Quand un problème est représenté pour être une en-

trée d'un tel solveur QCSP, les contraintes sont issues d'un ensemble de contraintes primitives du langage mais rarement un problème représentant un jeu à deux joueurs est exprimé comme une telle conjonction mais plutôt par une grande formule logique. Par exemple, dans un jeu fini à deux joueur sur n tours, le problème est exprimé avec une alternance de vérification des règles pour le joueur existentiel ($R_\exists$) et pour le joueur universel, l'adversaire, ($R_\forall$) et terminé par une vérification des conditions de victoire pour le joueur existentiel (VC) :

$$\begin{array}{l}\exists x_1 \forall y_1 \ldots \exists x_{n-1} \forall y_{n-1} \exists x_n \\ R_\exists(x_1) \wedge (R_\forall(x_1, y_1) \rightarrow \\ \quad (\ldots \quad (R_\exists(x_1, y_1, \ldots, x_{n-1}) \wedge \\ \qquad\quad (R_\forall(x_1, y_2, \ldots, x_{n-1}, y_{n-1}) \rightarrow \\ \qquad\quad (R_\exists(x_1, y_2, \ldots, x_{n-1}, y_{n-1}, x_n) \wedge \\ \qquad\quad VC(x_1, y_2, \ldots, x_{n-1}, y_{n-1}, x_n)))) \ldots))\end{array}$$

Pour transformer ce QCSP complexe en une entrée pour un $\wedge$-solveur QCSP, des symboles propositionnels existentiellement quantifiés sont introduits. Mais si cette transformation préserve les solutions du QCSP, elle change le nombre de nœuds explorés. La principale raison est que la propriété d'absorption du vrai par rapport à la disjonction n'est pas préservée dans le $\wedge$-solveur QCSP. Nous introduisons un nouvel outil, l'outil *coupure*, qui restaure cette propriété sans avoir à modifier le cœur du solveur. Ce nouvel outil est inspiré de la coupure de Prolog comme un outil sous la responsabilité du concepteur du QCSP pour élaguer des parties de l'espace de recherche dont le parcours est connu comme étant inutile par construction. La section 2 énonce les préliminaires indispensables. La section 3 présente notre nouvel outil. La section 4 discute de la gestion des quantificateurs universels vis-à-vis de la librairie CSP sous-jacente. La section 5 évalue notre solveur de QCSP, QuaCode, et l'impact de l'outil *coupure*. La section 6 dresse quelques perspectives.

2 Préliminaires

Le symbole $\mathcal{PS}$ représente les symboles propositionnels, le symbole $\exists$ représente le quantificateur existentiel et le symbole $\forall$ représente le quantificateur universel ($\overline{\exists} = \forall$ et $\overline{\forall} = \exists$). Le symbole $\wedge$ représente la conjonction logique, le symbole $\vee$ représente la disjonction logique, le symbole $\rightarrow$ représente l'implication logique, le symbole $\leftrightarrow$ représente l'équivalence logique, le symbole $\top$ représente ce qui est toujours vrai et le symbole $\bot$ représente ce qui est toujours faux. Le symbole $\equiv$ représente l'équivalence entre formules. Un QCSP est quintuplet $(\mathbf{V}, \text{ordre}, \text{quant}, \mathbf{D}, \mathbf{C})$: $\mathbf{V}$ est un ensemble de n variables, ordre est une bijection de $\mathbf{V}$ dans $[1..n]$, quant est une fonction de $\mathbf{V}$ dans $\{\exists, \forall\}$ (quant(v) dénote le quantificateur associé à la variable v), $\mathbf{D}$ est une fonction de $\mathbf{V}$ dans l'ensemble des domaines $\{D(v_1), \ldots, D(v_n)\}$ telle que, pour toute variable $v_i \in \mathbf{V}$, $D(v_i)$ en dénote son domaine, i.e. l'ensemble fini de toutes les valeurs possibles ($D(v)$ est le domaine associé à la variable v), $\mathbf{C}$ est un ensemble de contraintes. Si $v_{j_1}, \ldots, v_{j_m}$ sont les variables d'une contrainte $c_j \in \mathbf{C}$ alors la relation associée à c_j est un sous-ensemble du produit cartésien $D(v_{j_1}) \times \ldots \times D(v_{j_m})$. Dans ce qui suit, pour chaque $i \in [1..n]$, $q_{v_i} = \text{quant}(v_i)$ et $D_{v_i} = D(v_i)$.

Un QCSP $(\mathbf{V}, \text{ordre}, \text{quant}, \mathbf{D}, \mathbf{C})$ est généralement représenté par sa formule en logique du premier ordre $q_{v_1} v_1 \ldots q_{v_n} v_n \bigwedge_{c_j \in \mathbf{C}} c_j$ avec $v_1 \in D_{v_1}$, …, $v_n \in D_{v_n}$, ordre$(v_i) = i$, pour chaque $i \in [1..n]$. Avec cette représentation $q_{v_1} v_1 \ldots q_{v_n} v_n$ est appelé « lieur » (un lieur vide étant noté ε).

Algorithme 1 $dec_{\wedge\exists}$, fonction de décomposition d'un QCSP complexe en un QCSP équivalent sous la forme d'un $\wedge\exists$-QCSP.

Entrée: Une formule ϕ
Entrée: Un nombre n
Sortie: Une paire composée d'un lieur et d'une formule
  si $\phi = \top$ ou $\phi = \bot$ ou ϕ est une contrainte primitive **alors**
    retourner (ε, ϕ)
  sinon
    $\phi = [o_n \leftarrow (x \circ y)](\phi')$, avec $o_n \in \mathcal{PS}$, $\circ \in \{\wedge, \vee, \rightarrow\}$ et $x, y \in \mathcal{PS}$ ou une contrainte primitive
    $(Q, D) = dec_{\wedge\exists}(\phi', n+1)$
    retourner $(Q\exists o_n, (((x \circ y) \leftrightarrow o_n) \wedge D))$
  fin si

Un QCSP $Q\phi$ est considéré comme *complexe* si certaines de ses contraintes contiennent des connecteurs logiques sinon il est considéré comme *décomposé*. L'algorithme 1, qui spécifie la fonction $dec_{\wedge\exists}$ qui décompose un QCSP complexe en introduisant des symboles propositionnels existentiellement quantifiés intermédiaires, préserve la sémantique du QCSP : Si $(Q_\exists, D) = dec_{\wedge\exists}(C, 1)$ alors $QC \equiv QQ_\exists D$. Le dual d'un QCSP (primal) en est sa négation : $\overline{q_{v_1}} v_1 \ldots \overline{q_{v_n}} v_n \bigvee_{c_j \in \mathbf{C}} \neg c_j$ avec $v_1 \in D_{v_1}$, …, $v_n \in D_{v_n}$. Le dual d'un QCSP complexe peut aussi être décomposé mais par introduction de symboles propositionnels universellement quantifiés suivant trois modifications de la fonction $dec_{\wedge\exists}$ en une fonction $dec_{\vee\forall}$: les contraintes primitives sont niées, l'appel récursif est sur $dec_{\vee\forall}$ et la dernière instruction est remplacée par **retourner** $(Q\forall o_n, (((x \circ y) \leftrightarrow o_n) \vee D))$. Cette décomposition selon la fonction $dec_{\vee\forall}$ préserve aussi la sémantique du QCSP d'origine : Si $(Q_\forall, D) =$

$dec_{\vee\forall}(C,1)$ alors $\neg QC \equiv \overline{Q}Q_{\forall}D$.

L'ensemble $\mathcal{T}(Q)$ avec $v_1 \in D_{v_1}$, ..., $v_n \in D_{v_n}$, $Q = q_{v_1}v_1 \dots q_{v_n}v_n$ est l'ensemble des arbres tels que :

- chaque nœud feuille est étiqueté avec le symbole $\Box$ et à la profondeur n,
- chaque nœud interne à la profondeur k, $0 \leq k < n$, est étiqueté avec la variable v_{k+1},
- chaque arc reliant un nœud de profondeur k à l'un de ses fils est étiqueté par un élément de D_{v_k},
- toutes les étiquettes des arcs reliant un nœud à l'un de ses fils sont différentes.

Soit $(\mathbf{V}, \text{ordre}, \text{quant}, \mathbf{D}, \mathbf{C})$ un QCSP tel que $\mathbf{V} = \{v_1, \dots, v_n\}$, avec $v_1 \in D_{v_1}$, ..., $v_n \in D_{v_n}$. Une stratégie est un arbre de $\mathcal{T}(q_{v_1}v_1 \dots q_{v_n}v_n)$ tel que

- chaque nœud étiqueté par une variable existentiellement quantifiée admet un unique fils et
- chaque nœud étiqueté par une variable universellement quantifiée dont le domaine est de taille k admet k nœuds fils.

Un scénario est une séquence d'étiquettes $val_1, \dots, val_n$ sur un chemin $(v_1, val_1), \dots, (v_n, val_n)$, $val_i \in D_{v_i}$ pour tout i, $1 \leq i \leq n$, d'un arbre $\mathcal{T}(q_{v_1}v_1 \dots q_{v_n}v_n)$. Un scénario $val_1, \dots, val_n$ pour un QCSP $(\mathbf{V}, \text{ordre}, \text{quant}, \mathbf{D}, \mathbf{C})$ tel que $\mathbf{V} = \{v_1, \dots, v_n\}$ est un scénario gagnant si $(\bigwedge_{1 \leq i \leq n} v_i = val_i) \wedge (\bigwedge_{c_j \in \mathbf{C}} c_j)$ est vrai ; un tel scénario correspond à l'instanciation complète $v_1 = val_1$, ..., $v_n = val_n$; c'est un scénario gagnant si l'instanciation satisfait toutes les contraintes. Une stratégie est une stratégie gagnante si tous les scénarios sont des scénarios gagnants. S'il n'y a pas de quantificateur, la stratégie $\Box$ est toujours une stratégie gagnante.

Nous pouvons donner une sémantique plus intuitive pour les QCSP : un QCSP $\forall xQC$ avec $x \in D_x$ admet une stratégie gagnante si et seulement si, pour tout $val \in D_x$, $Q(C \wedge (x = val))$ admet une stratégie gagnante et un QCSP $\exists xQC$ avec $x \in D_x$ admet une stratégie gagnante si et seulement si, pour au moins un $val \in D_x$, $Q(C \wedge (x = val))$ admet une stratégie gagnante.

Les stratégies gagnantes des QCSP complexes sont aisément accessibles à partir de stratégies gagnantes des QCSP décomposés correspondants en remplaçant les sous arbres introduits par les symboles propositionnels existentiellement quantifiés intermédiaires par des nœuds feuilles étiquetés par $\Box$.

L'algorithme 2 présente la structure d'un algorithme de recherche quantifié pour un QCSP complexe décomposé selon l'application de l'algorithme $dec_{\wedge\exists}$.

Algorithme 2 Un algorithme de recherche quantifié pour solveur de $\wedge$-QCSP

Entrée: Un QCSP $Q\phi$
Sortie: **vrai** si le QCSP $Q\phi$ admet au moins une stratégie gagnante et **faux** sinon
$C := dec_{\wedge\exists}(\phi, 1)$
$pileRetourArrière := \emptyset$
tant que *true* **faire**
selon $atteintPointFixe(C)$ **faire**
cas *failure*
retour sur le dernier choix existentiel (x, k) de $pileRetourArrière$
si aucune **alors retourner** faux
sinon ajouter à C la contrainte $(x = k)$
cas *succes*
retour sur le dernier choix universel (x, k) de $pileRetourArrière$
si aucune **alors retourner** vrai
sinon ajouter à C la contrainte $(x = k)$
cas *branche*
sélectionner la variable non instanciée suivante x
pour tout $k \in D(x)$ empiler (x, k) dans $pileRetourArrière$
sélectionner le premier choix (x, k) de $pileRetourArrière$
ajouter à C la contrainte $(x = k)$
fin selon
fin tant que

3 L'outil *coupure*

De nos jours les solveurs de contraintes quantifiées sont basés sur l'utilisation d'une file de contraintes considérée comme une conjonction de contraintes primitives. Le cadre QCSP est un cadre puissant pour représenter des problèmes avec adversaires mais ces problèmes sont rarement décrits sous la forme d'une conjonction n-aire de contraintes primitives. Ces problèmes sont représentés avec des contraintes complexes incluant des disjonctions et des implications [1] Pour traiter un QCSP avec un solveur QCSP moderne, les problèmes doivent être transformés : les contraintes complexes sont décomposées grâce à un algorithme de décomposition (cf. la section 2).

Le QCSP $q_{x_1}x_1 \dots q_{x_i}x_i q_{x_{i+1}}x_{i+1} \dots q_{x_n}x_n$ $R(x_1, \dots, x_i) \vee G(x_1, \dots, x_n)$ est, par exemple, décomposé en le QCSP

$$q_{x_1}x_1 \dots q_{x_i}x_i q_{x_{i+1}}x_{i+1} \dots q_{x_n}x_n \\ \exists o_1 \exists o'_1 \exists o_2 \dots o_m \exists o'_2 \dots o'_p \\ (o_1 \vee o'_1) \wedge R_{dec}(x_1, \dots, x_i) \wedge G_{dec}(x_1, \dots, x_n)$$

avec

1. et des quantificateurs nichés dans les formules mais ceci est hors de la portée de cet article.

$$\begin{cases} (R_{dec}(x_1,\ldots,x_i),o_1,\ldots,o_m) \\ = \quad dec_{\wedge\exists}(R(x_1,\ldots,x_i),1) \text{ et} \\ (G_{dec}(x_1,\ldots,x_n),o'_1,\ldots,o'_p) \\ = \quad dec_{\wedge\exists}(G(x_1,\ldots,x_n),1) \end{cases}$$

Même si les deux QCSP sont équivalents, le second est bien plus complexe à résoudre car de nombreuses instances satisfiables sont explorées durant la recherche. Ces branches sont alors combinées (au niveau des variables universellement quantifiées) pour construire des stratégies gagnantes. Une propriété de la logique sous-jacente très importante pour l'efficacité est perdue dans un $\wedge$-solveur : la propriété d'absorption du $\top$ par rapport à la disjonction i.e. si un argument d'une disjonction n-aire est vrai alors il est inutile de calculer les autres arguments, la disjonction est vraie. Si $R(x_1,\ldots,x_i)$ est vrai pour une instanciation quelconque alors il n'est pas nécessaire de résoudre $q_{x_{i+1}}x_{i+1}\ldots q_{x_n}x_nG(x_1,\ldots,x_n)$ pour cette instanciation mais le $\wedge$-solveur le fait.

Exemple 1 *Le jeu de Nim est un jeu à deux joueurs qui se joue avec un tas de pièces ou d'allumettes. Le but du jeu est d'être celui qui se saisira de la dernière allumette. Chaque joueur peut prendre de une à trois allumettes. Avec la variante de Fibonacci, le minimum est d'une allumette et le maximum pour le premier tour est du nombre initial d'allumettes moins une. Puis chaque joueur peut prendre d'une jusqu'au double d'allumettes que son adversaire à pris au tour précédent. Le joueur existentiel débute. Avec un nombre pair n d'allumettes, le QCSP est le suivant (x_i est le nombre d'allumettes choisies au tour i) :* $R_\exists(1) = \top$ *(Le joueur existentiel choisit entre 1 et* $n-1$ *allumettes) et pour tout i,* $1 < i \leq \frac{n}{2}$, $R_\exists(i) = (x_i \leq 2 * y_{i-1}) \wedge (x_i + \Sigma_{1\leq j<i}(x_j + y_j) \leq n)$ *et pour tout i,* $1 \leq i \leq \frac{n}{2}$, $R_\forall(i) = (y_i \leq 2{*}x_i) \wedge (\Sigma_{1\leq j\leq i}(x_j{+}y_j) \leq n)$ *(chaque joueur prend d'une jusqu'au double d'allumettes que son adversaire à pris au tour précédent) :*

$\exists x_1 \forall y_1 \ldots \exists x_{n-1} \forall y_{n-1}$
$R_\exists(1) \wedge (R_\forall(1) \rightarrow (R_\exists(2) \wedge (R_\forall(2) \rightarrow$
$(\ldots(R_\forall(n-1) \rightarrow \bot)))))$

avec $x_1, y_1, \ldots, x_{n-1}, y_{n-1} \in [1..n-1]$.

Pour $n = 4$*, nous obtenons le QCSP :*

$\exists x_1 \forall y_1 \exists x_2 \forall y_2$
$(((y_1 \leq 2 * x_1) \wedge ((x_1 + y_1) \leq 4)) \rightarrow$
$(((x_2 \leq 2 * y_1) \wedge ((x_1 + y_1 + x_2) \leq 4)) \wedge$
$(((y_2 \leq 2 * x_2) \wedge ((x_1 + y_1 + x_2 + y_2) \leq 4)) \rightarrow \bot)))$
avec $x_1, y_1, x_2, y_2 \in \{1, 2, 3\}$

Le dual du QCSP est le QCSP suivant :

$\forall x_1 \exists y_1 \forall x_2 \exists y_2$
$(((y_1 \leq 2 * x_1) \wedge ((x_1 + y_1) \leq 4)) \wedge$
$(((x_2 \leq 2 * y_1) \wedge ((x_1 + y_1 + x_2) \leq 4)) \rightarrow$
$(((y_2 \leq 2 * x_2) \wedge ((x_1 + y_1 + x_2 + y_2) \leq 4)) \wedge \top)))$
avec $x_1, y_1, x_2, y_2 \in \{1, 2, 3\}$

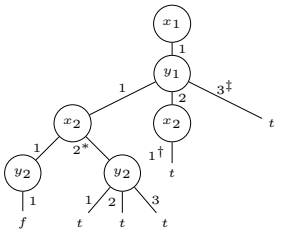


FIGURE 1 – Arbre de recherche pour le problème (primal) original

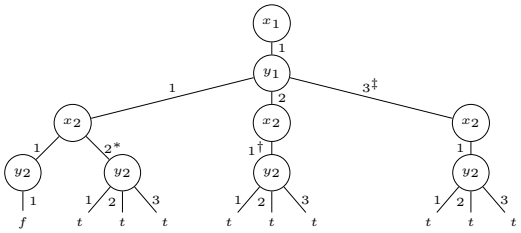


FIGURE 2 – Arbre de recherche pour le primal décomposé selon la $\wedge\exists$-décomposition

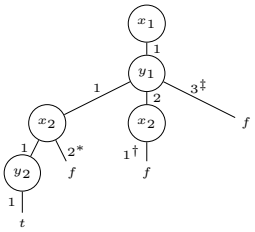


FIGURE 3 – Arbre de recherche pour le dual décomposé selon la $\wedge\exists$-décomposition

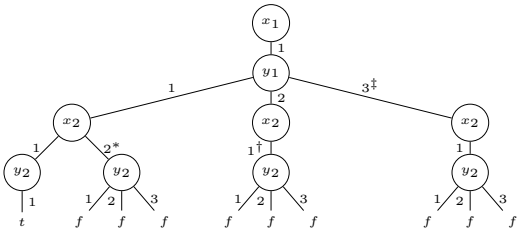


FIGURE 4 – Arbre de recherche pour le dual décomposé selon la $\vee\forall$-décomposition

Pour l'instanciation $(x_1 = 1)$, $(y_1 = 1)$, $(x_2 = 1)$*, le QCSP est réduit à* $\forall y_2(((y_2 \leq 2) \wedge (y_2 \leq 1)) \rightarrow \bot) \equiv \bot$ *avec* $y_2 \in \{1, 2, 3\}$ *et pour l'instanciation* $(x_1 = 1)$, $(y_1 = 1)$, $(x_2 = 2)$ *(∗ de la figure 1), le QCSP est*

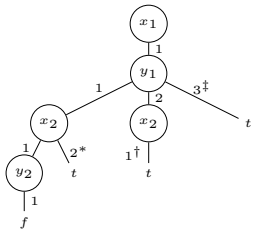


FIGURE 5 – Arbre de recherche pour le QCSP primal avec coupure

réduit à $\forall y_2(((y_2 \leq 4) \wedge (y_2 \leq 0)) \rightarrow \bot) \equiv \top$ *avec* $y_2 \in \{1,2,3\}$.

Pour l'instanciation $(x_1 = 1)$, $(y_1 = 2)$, $(x_2 = 1)$ *(† de la figure 1), le QCSP est réduit à* $\forall y_2(((y_2 \leq 2) \wedge (y_2 \leq 0)) \rightarrow \bot) \equiv \top$ *avec* $y_2 \in \{1,2,3\}$.

Pour l'instanciation $(x_1 = 1)$, $(y_1 = 3)$ *(‡ de la figure 1), le QCSP est réduit à* $\exists x_2 \forall y_2 \lambda \equiv \top$ *avec* $x_2, y_2 \in \{1,2,3\}$ *et* $\lambda = ((3 \leq 2) \rightarrow (((x_2 \leq 6) \wedge (x_2 \leq 0)) \wedge (((y_2 \leq 2 * x_2) \wedge ((x_2 + y_2) \leq 0)) \rightarrow \bot)))$. *Mais en fait,* $\forall x_2 \forall y_2 \lambda \equiv \top$ *avec* $x_2, y_2 \in \{1,2,3\}$. *Sous l'instanciation partielle* $*$, † *et* ‡, *toutes les branches sont vraies.*

Par construction, la propriété d'absorption de $\bot$ par rapport à la conjonction est assurée par un solveur (Q)CSP : si une contrainte est fausse, puisque la file est une conjonction n-aire, il n'est pas nécessaire de calculer les autres contraintes et le solveur revient en arrière. Dans un $\wedge$-solveur pour QCSP, la propriété d'absorption de $\top$ par rapport à la disjonction n'est plus assurée. Même si pour une instanciation quelconque la conjonction de contraintes $dec_{\wedge\exists}(R(x_1, \ldots, x_i))$ est vrai (et donc $((o_1 \vee o_1') \leftrightarrow \top)$ est aussi vrai), le $\wedge$-solveur QCSP tentera de résoudre le QCSP $q_{x_{i+1}} x_{i+1} \ldots q_{x_n} x_n dec(G(x_1, \ldots, x_n))$ pour cette instanciation. En d'autres termes, pour détecter un succès, dans un $\wedge$-solveur QCSP, toutes les contraintes doivent être satisfaites et tant qu'il demeure des variables non instanciées ou des contraintes non résolues, l'algorithme de recherche quantifié 2 ne peut pas décider.

Exemple 2 *La décomposition selon l'algorithme* $\wedge\exists$*-dec du QCSP de l'exemple 1 est le QCSP suivant :*

$\exists x_1 \forall y_1 \exists x_2 \forall y_2 \exists o_1 \exists o_2$
$(((y_1 \leq 2 * x_1) \wedge ((x_1 + y_1) \leq 4)) \rightarrow o_1) \wedge$
$(o_1 \leftrightarrow (((x_2 \leq 2 * y_1) \wedge ((x_1 + y_1 + x_2) \leq 4)) \wedge o_2)) \wedge$
$(o_2 \leftrightarrow (((y_2 \leq 2 * x_2) \wedge ((x_1 + y_1 + x_2 + y_2) \leq 4)) \rightarrow \bot))$
avec $x_1, y_1, x_2, y_2 \in \{1,2,3\}, o_1, o_2 \in$ **BOOL**

La figure 2 montre l'arbre de recherche pour le QCSP ci-dessus :

Sous les instanciations partielles $*$, † *et* ‡, *toutes les branches sont vraies mais la propriété d'absorption du* $\top$ *par rapport à la disjonction n'est pas appliquée et les branches ne sont pas élaguées.*

La décomposition du dual selon l'algorithme $\wedge\exists$*-dec du QCSP de l'exemple 1 est le QCSP suivant :*

$\forall x_1 \exists y_1 \forall x_2 \exists y_2 \exists o_1 \exists o_2$
$(((y_1 \leq 2 * x_1) \wedge ((x_1 + y_1) \leq 4)) \wedge o_1) \wedge$
$(o_1 \leftrightarrow (((x_2 \leq 2 * y_1) \wedge ((x_1 + y_1 + x_2) \leq 4)) \rightarrow o_2)) \wedge$
$(o_2 \leftrightarrow (((y_2 \leq 2 * x_2) \wedge ((x_1 + y_1 + x_2 + y_2) \leq 4)) \wedge \top))$
avec $x_1, y_1, x_2, y_2 \in \{1,2,3\}, o_1, o_2 \in$ **BOOL**

La figure 3 montre l'arbre de recherche pour le QCSP ci-dessus :

Sous les instanciations partielles $*$, † *et* ‡, *toutes les branches sont fausses et la propriété d'absorption du* $\bot$ *par rapport à la conjonction est appliquée, ainsi les branches sont élaguées.*

De la même manière et aussi par construction, la propriété d'absorption du $\top$ par rapport à la disjonction est aussi assurée par un solveur dual pour un QCSP basé sur une file considérée comme une disjonction n-aire, mais dans ce cas, c'est la propriété d'absorption du $\bot$ par rapport à la conjonction qui n'est plus assurée. En d'autres termes, pour détecter un échec, dans un solveur dual pour QCSP, toutes les contraintes doivent être violées et tant qu'il demeure des variables non instanciées ou des contraintes non violées, l'algorithme de recherche quantifié 2 ne peut pas décider.

Exemple 3 *La décomposition* $\vee\forall$ *du dual du QCSP de l'exemple 1 est le QCSP suivant :*

$\forall x_1 \exists y_1 \forall x_2 \exists y_2 \forall o_1 \forall o_2$
$(((y_1 \leq 2 * x_1) \wedge ((x_1 + y_1) \leq 4)) \wedge o_1) \vee$
$(o_1 \leftrightarrow ((\neg(x_2 \leq 2 * y_1) \vee \neg((x_1 + y_1 + x_2) \leq 4)) \vee o_2)) \vee$
$(o_2 \leftrightarrow (((y_2 \leq 2 * x_2) \wedge ((x_1 + y_1 + x_2 + y_2) \leq 4)) \wedge \top))$
avec $x_1, y_1, x_2, y_2 \in \{1,2,3\}, o_1, o_2 \in$ **BOOL**

Le figure 4 montre l'arbre de recherche du QCSP ci-dessus.

Sous les instanciations partielles $*$, † *et* ‡, *toutes les branches sont fausses mais la propriété d'absorption de* $\bot$ *par rapport à la conjonction n'est pas appliquée et les branches ne sont pas élaguées.*

Quand toutes les branches du sous-arbre du dual sont fausses, nous pouvons en toute sécurité élaguer ces branches dans le problème primal car elles correspondent à des branches succès. Dans le but d'élaguer ces branches, nous introduisons un outil, la « coupure », avec la sémantique suivante :

Définition 1 *L'outil coupure est défini, pour un ensemble de contraintes* S *et une contrainte* e *comme*

coupure(S, e) *: si* $\neg(\bigwedge_{c \in S} c \rightarrow e)$ *est vrai alors le QCSP courant est résolu.*

L'outil *coupure* est un outil sous la responsabilité du concepteur du QCSP puisque si la coupure est vraie il est attendu que l'ensemble des branches du QCSP courant soient vraies sans la moindre vérification. L'outil *coupure* est inutile pour les CSPs puisque les solveurs CSP arrêtent au premier succès.

Exemple 4 *Nous ajoutons au* $\wedge\exists$*-QCSP de l'exemple 2 les deux nouvelles coupures :*

$$\begin{array}{l} \mathit{coupure}(\{\}, ((y_1 \leq 2 * x_1) \wedge ((x_1 + y_1) \leq 4))) \\ \mathit{coupure}(\quad \{(y_1 \leq 2 * x_1), ((x_1 + y_1) \leq 4), (x_2 \leq 2 * y_1), \\ \qquad ((x_1 + y_1 + x_2) \leq 4)\}, \\ \qquad ((y_2 \leq 2 * x_2) \wedge ((x_1 + y_1 + x_2 + y_2) \leq 4))) \end{array}$$

La première coupure ôte le sous-arbre sous ‡ (cf. figure 2) :

$$[x_1 \leftarrow 1][y_1 \leftarrow 3](\neg((y_1 \leq 2 * x_1) \wedge ((x_1 + y_1) \leq 4)))$$

est vrai. La seconde coupure ôte les sous-arbres sous $*$ *et* † *(cf. figure 2) : Soit* $S = (y_1 \leq 2 * x_1) \wedge ((x_1 + y_1) \leq 4) \wedge (x_2 \leq 2 * y_1) \wedge ((x_1 + y_1 + x_2) \leq 4)$ *et* $e = ((y_2 \leq 2 * x_2) \wedge ((x_1 + y_1 + x_2 + y_2) \leq 4))$.

(*) $[x_1 \leftarrow 1][y_1 \leftarrow 1][x_2 \leftarrow 2](S)$ *est vrai et* $[x_1 \leftarrow 1][y_1 \leftarrow 1][x_2 \leftarrow 2](e)$ *est faux puisque* $y_2 \geq 0$ *donc* $[x_1 \leftarrow 1][y_1 \leftarrow 1][x_2 \leftarrow 2](\neg(S \rightarrow e))$ *est vrai.*

(†) $[x_1 \leftarrow 1][y_1 \leftarrow 2][x_2 \leftarrow 1](S)$ *est vrai et* $[x_1 \leftarrow 1][y_1 \leftarrow 2][x_2 \leftarrow 1](e)$ *est faux puisque* $y_2 \geq 0$ *donc* $[x_1 \leftarrow 1][y_1 \leftarrow 2][x_2 \leftarrow 1](\neg(S \rightarrow e))$ *est vrai.*

Dans tous les cas, $\sigma(\neg(S \rightarrow e))$ *est vrai pour une instanciation* σ *signifie que le joueur universel essaye de tricher pour continuer à joueur (et donc perd automatiquement). La figure 5 montre l'arbre de recherche pour le* $\wedge\exists$*- QCSP augmenté de ces deux nouvelles contraintes. Le nombre de branches est de 4 au lieu de 10 pour l'arbre de recherche du QCSP primal (cf. figure 2) ce qui est un gain important.*

Tout jeu fini à deux joueurs peut être représenté comme un QCSP complexe :

$$\begin{array}{l} Q\phi = \exists x_1 \forall y_1 \ldots \exists x_{n-1} \forall y_{n-1} \exists x_n \\ R_\exists(x_1) \wedge (R_\forall(x_1, y_1) \rightarrow \\ \quad (\ldots \quad (R_\exists(x_1, y_1, \ldots, x_{n-1}) \wedge \\ \qquad (R_\forall(x_1, y_1, \ldots, x_{n-1}, y_{n-1}) \rightarrow \\ \qquad (R_\exists(x_1, y_1, \ldots, x_{n-1}, y_{n-1}, x_n) \wedge \\ \qquad VC(x_1, y_1, \ldots, x_{n-1}, y_{n-1}, x_n)))) \ldots)) \end{array}$$

Les variables $x_1 \in D_{x_1}, y_1 \in D_{y_1}, \ldots, x_{n-1} \in D_{x_{n-1}}, y_{n-1} \in D_{y_{n-1}}, x_n \in D_{x_n}$ représentent les mouvements, n le nombre de tours, $R_\exists(x_1, y_1, \ldots, x_i)$ les règles à respecter pour le joueur existentiel au tour i, $R_\forall(x_1, y_1, \ldots, x_{i-1}, y_i)$ les règles pour le joueur universel au tour i et $VC(x_1, y_1, \ldots, x_n)$ les conditions de victoire pour le joueur existentiel. Le joueur existentiel gagne si le QCSP est vrai, sinon, c'est le joueur qui gagne. Un joueur gagne immédiatement si son adversaire triche.

Le théorème suivant exprime qu'avec l'outil *coupure*, le $\wedge$-solveur pour le QCSP obtenu par une décomposition selon l'algorithme $dec_{\wedge\exists}$ calcule les mêmes stratégies gagnantes que le $\wedge$-solveur pour QCSP sans la coupure mais qu'avec l'outil de coupure le $\wedge$-solveur pour QCSP calcule le même arbre de recherche que pour le QCSP complexe (qui est bien plus petit que l'arbre de recherche pour le QCSP basé sur une décomposition selon l'algorithme $dec_{\wedge\exists}$).

Théoreme 1 *Soit* $Q\phi$ *le QCSP complexe suivant spécifiant un jeu à deux joueurs :*

$$\begin{array}{l} Q\phi = \exists x_1 \forall y_1 \ldots \exists x_{n-1} \forall y_{n-1} \exists x_n \\ R_\exists(x_1) \wedge (R_\forall(x_1, y_1) \rightarrow \\ \quad (\ldots \quad (R_\exists(x_1, y_1, \ldots, x_{n-1}) \wedge \\ \qquad (R_\forall(x_1, y_1, \ldots, x_{n-1}, y_{n-1}) \rightarrow \\ \qquad (R_\exists(x_1, y_1, \ldots, x_{n-1}, y_{n-1}, x_n) \wedge \\ \qquad VC(x_1, y_1, \ldots, x_{n-1}, y_{n-1}, x_n)))) \ldots)) \end{array}$$

avec $x_1 \in D_{x_1}, y_1 \in D_{y_1}, \ldots, x_{n-1} \in D_{x_{n-1}}, y_{n-1} \in D_{y_{n-1}}, x_n \in D_{x_n}$.

Soit $(Q_\exists, \Delta) = dec_{\wedge\exists}(\phi, 1)$ *et l'ensemble*

$$\begin{array}{l} \Sigma = \{\mathit{coupure}(\{R_\exists(x_1), \ldots, R_\exists(x_1, y_1, \ldots, y_{i-1}, x_i)\} \\ \quad \cup \{R_\forall(x_1, y_1), \ldots, R_\forall(x_1, y_1, \ldots, x_{i-1}, y_{i-1})\}, \\ \quad R_\forall(x_1, y_1, \ldots, y_{i-1}, x_i, y_i)\ ) \mid 1 \leq i \leq n\ \} \end{array}$$

de coupures (avec $R_\forall(x_1, y_1, \ldots, y_{i-1}, x_i, y_i) = \bot$*). Les propositions suivantes sont vérifiées :*

— *Soit s une stratégie pour* $QQ_\exists$*. s est une stratégie gagnante pour* $QQ_\exists\Delta$ *si et seulement si s est une stratégie gagnante pour* $QQ_\exists(\Delta \wedge \Sigma)$.

— *Soit T un arbre de recherche pour Q. T est un arbre de recheche de* $Q\phi$ *si et seulement si T est un arbre de recherche pour* $QQ_\exists(\Delta \wedge \Sigma)$.

Le théorème présenté ci-dessus est une application de l'outil *coupure* pour un QCSP spécifiant un jeu à deux joueurs. Mais cet outil peut être utilisé pour n'importe quelle expression logique qui assure le succès du QCSP lorsqu'elle devient vraie. Le principal avantage de cet outil est qu'il peut être utilisé dans n'importe quel solveur du moment même où l'on peut décrire ses propres propagateurs. D'un autre côté, la responsabilité de sa juste utilisation incombe au concepteur du QCSP. Dans [5], les auteurs décrivent une approche permettant de restituer l'absorption de la contrainte *ou*. Cette approche a le même impact que l'outil *coupure* pour le traitement de la contrainte *ou* mais il nécessite la modification profonde du cœur du solveur.

En effet, il est nécessaire de représenter les liens entre les contraintes primitives sous forme d'arbre mais aussi de modifier les mécanismes internes permettant de décider si une contrainte doit être propagée au non. Cette approche est en contradiction avec l'hypothèse initiale de cet article qui veut que nous ne touchions pas au cœur du solveur. De plus l'outil *coupure* n'est pas restreint au traitement de la contrainte *ou* mais peut être utilisé pour des contraintes plus complexes.

4 Discussion

Pour étudier l'impact de l'outil *coupure* sur la résolution d'instances QCSP, nous avons développé QuaCode un solveur QCSP construit au-dessus de GeCode [2]. Ainsi, toute contrainte présente dans GeCode est disponible dans QuaCode. QuaCode implémente l'algorithme 2 pour chercher une stratégie gagnante. Même si cet algorithme est complet et permet de résoudre toute instance QCSP, il instancie inutilement certaines variables universelles augmentant considérablement le temps de résolution. Prenons l'exemple suivant : $\exists x \exists y \forall z (y + z \leq x)$ avec $x, y, z \in \{0, 1, 2\}$.

En utilisant l'algorithme 2 nous obtenons l'arbre de recherche de la figure 6. Cet arbre peut facilement être réduit. En effet, si x est instancié à 0 ou 1, la propagation permet de contracter le domaine de z en retirant 2. Or z est une variable universelle, cela conduit donc à un échec. L'algorithme 2 ne détecte pas cet échec dès que x est instancié à 0 ou 1, mais doit attendre que toutes les variables de la branche soient instanciées.

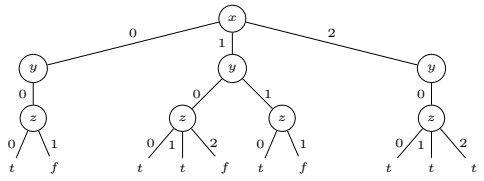


FIGURE 6 – Arbre de recherche obtenu avec l'algorithme 2

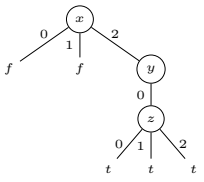


FIGURE 7 – Arbre de recherche optimisé.

2. `http://www.gecode.org`

Il y a deux raisons permettant à un solveur CSP de retirer une valeur d'un domaine d'une variable : soit durant l'étape de propagation suite à la contraction d'un domaine par une contrainte, soit lors de l'étape de branchement par instanciation d'une variable. Dans un QCSP, l'instanciation d'une variable est réalisé de la même manière que dans un CSP (le domaine de la variable choisie est réduit). Cela n'est pas vrai pour la contraction d'un domaine pendant la phase de propagation. En effet, même si la contraction d'une variable existentielle s'effectue de la même manière, toute contraction d'une variable universelle doit conduire à un échec de la branche courante de l'arbre de recherche. En surveillant les variables quantifiées universellement, il est possible de détecter ces échecs dès qu'une variable universelle est contractée. L'arbre de recherche correspondant sera donc plus petit comme le montre la figure 7.

Pour implémenter ce mécanisme de surveillance dans QuaCode sans changer le cœur de la bibliothèque GeCode, nous avons développé une *contrainte de surveillance*. Cette contrainte surveille toute modification du domaine d'une variable survenant pendant la phase de propagation. Pour implémenter une telle contrainte, nous avons besoin d'une variable existentielle supplémentaire x_{aux} insérée, dans le lieur, juste après la variable universelle $x_\forall$ à surveiller. Une contrainte de surveillance est définie comme suit :

$watch(x_\forall, x_{aux}) =$
si $D(x_\forall) = D(x_{aux})$ **ne rien faire**
si $D(x_\forall) \subset D(x_{aux})$ **échec**
sinon $D(x_\forall) \leftarrow D(x_{aux})$

Il est donc nécessaire de poster autant de contraintes de surveillance qu'il y a de variables universelles dans le problème. Les variables x_{aux} ne peuvent pas être modifiées lors de la phase de propagation car elles n'apparaissent pas dans les contraintes du problème (sauf dans la contrainte de surveillance). Ce n'est que lors d'un branchement qu'une variable x_{aux} peut être modifiée. La variable $x_\forall$ associée sera mise à jour lors de l'étape de propagation qui suivra. Si $x_\forall$ est est modifié durant la phase de propagation, alors la contrainte de surveillance provoque un échec. Les variables supplémentaires ajoutées n'augmentent pas la taille de l'espace de recherche. En effet, l'algorithme de recherche ne voit que les variables x_{aux} et non les variables initiales du problème, la taille de l'espace de recherche reste inchangé.

5 Expérimentations

5.1 Comparaison de QuaCode avec l'état de l'art

Pour comparer QuaCode avec les principaux solveurs QCSP de l'état de l'art, nous avons utilisé deux

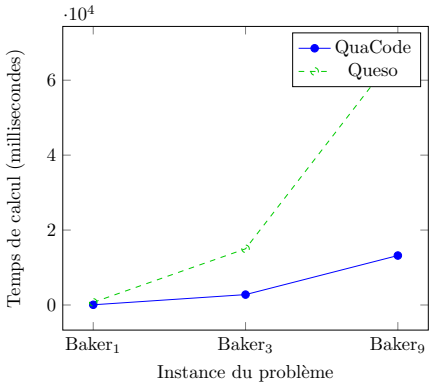
problèmes classiques : le problème du boulanger et le jeu MatrixGame. Nous avons réalisé nos expérimentations sur un ordinateur sous Linux équipé d'un Intel i5 à 1.8GHz et 4GB de RAM. Le temps maximum pour la résolution d'une instance est fixé à une heure.

Le problème du boulanger consiste à aider un boulanger souhaitant acquérir quatre poids distincts à choisir dans l'intervalle $\{1, \ldots, 40\}$ kg, lui permettant de peser n'importe quelle quantité entière de farine dans l'intervalle $\{1, \ldots, 40\}$ kg en utilisant une balance de Roberval. Pour la pesée, chaque poids peut être placé de n'importe quel côté de la balance ou ne pas être utilisé. Ce problème peut modélisé par le QCSP suivant : sachant que $w_1, w_2, w_3, w_4, f \in \{1, \ldots, 40\}$, $c_1, c_2, c_3, c_4 \in \{-1, 0, 1\}$, alors $\exists w_1 \exists w_2 \exists w_3 \exists w_4 \forall f \exists c_1 \exists c_2 \exists c_3 \exists c_4 (\sum_{i=1}^{4} w_i.c_i = f)$.

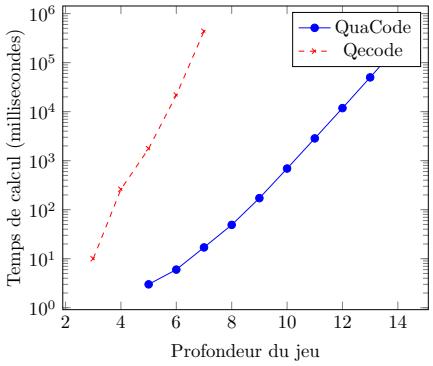
Le problème du boulanger admet une unique stratégie gagnante $\{1, 3, 9, 27\}$ (modulo les permutations des variables w_i). La première stratégie gagnante est trouvée rapidement quelque soit le solveur. Nous avons donc construit trois instances plus difficiles à résoudre en éliminant les premières stratégies gagnantes. L'instance Baker_1 correspond à l'instance initiale. L'instance Baker_3 empêche le solveur de trouver une stratégie gagnante où $w_1 = 1$, ainsi, la première stratégie gagnante sera $\{3, 1, 9, 27\}$. L'instance Baker_9 est construite de sorte que la première stratégie gagnante soit $\{9, 1, 3, 27\}$. Sur ce problème, nous comparons QuaCode et Queso [3], un solveur QCSP en Java écrit par P. Nightingale. Queso n'est plus maintenu, mais son code source est toujours disponible sur le site de l'auteur. Les résultats de la figure 8(a) montrent que QuaCode trouve la première stratégie gagnante plus rapidement que Queso quelque soit l'instance. Sur l'instance Baker_9, QuaCode est cinq fois plus rapide que Queso. Cela illustre que même si QuaCode n'est pas un solveur QCSP construit de zéro, les contraintes de GeCode, les contraintes de surveillance des variables universelles ainsi qu'un algorithme de recherche quantifié simple sont suffisant pour rivaliser avec un solveur ad'hoc.

Le jeu MatrixGame est un jeu à deux joueurs de d tours. Il est joué sur une matrice de 0/1 de taille 2^d. À chaque tour, le $\exists$-joueur coupe horizontalement la matrice en deux et décide si il faut conserver la partie haute ou basse. Puis, le $\forall$-joueur coupe la matrice verticalement et décide si il faut conserver la partie gauche ou droite. Le $\exists$-joueur gagne si la dernière case contient 1.

Le jeu MatrixGame peut aussi être modélisé par un QCSP. Pour ce problème, nous comparons QuaCode avec Qecode [4], un solveur QCSP/QCSP+ également basé sur GeCode. Pour résoudre un QCSP+, Qecode construit une succession de CSPs où toutes les variables sont quantifiées existentiellement. L'algorithme de recherche de solution de GeCode est appelé pour résoudre chaque CSP et les résultats sont récupérés et analysés par Qecode. Qecode connecte ensuite tous les résultats pour résoudre le QCSP+. Bien que le jeu MatrixGame soit souvent modélisé par un QCSP+, il n'y a aucune règle à vérifier entre chaque tour de jeu. Les contraintes dépendent de toutes les variables du problème, et donc aucune des spécificités des QCSP+ n'est nécessaire pour ce problème. Il peut donc être parfaitement modélisé par un QCSP. Le jeu MatrixGame est fourni avec Qecode mais pas avec Queso, nous n'avons donc pas retenu ce dernier pour cette expérience.



(a) Problème du boulanger



(b) MatrixGame

FIGURE 8 – Temps de calculs.

La figure 8(b) compare les temps de calcul de QuaCode et Qecode sur le jeu MatrixGame. Nous observons que QuaCode est systématiquement plus rapide que Qecode. Lorsque la profondeur du jeu dépasse 7, la différence dépasse le facteur mille. Comme les deux solveurs sont basés sur GeCode, le gain d'efficacité s'explique par l'utilisation des contraintes de sur-

3. `http://pn.host.cs.st-andrews.ac.uk/`
4. `http://www.univ-orleans.fr/lifo/software/qecode/`

veillance des variables universelles. En effet, Qecode construit plusieurs problèmes relâchés où les quantificateur universels sont remplacés par des existentiels. Pendant la résolution d'un des problèmes relâchés, le solveur agit comme une boîte noire sans aucune connaissance des quantificateurs originaux. Le solveur n'est donc pas capable de détecter rapidement les échecs dûs aux variables universelles et doit explorer tout le sous-arbre de recherche pour conclure à un échec. Au contraire, QuaCode coupe ces branches au plus tôt grâce aux contraintes de surveillance. Plus l'instance est grosse, plus le gain sera important car le sous-arbre évité sera plus important.

Les résultats expérimentaux sur ces deux problèmes montrent l'efficacité de QuaCode. Pourvu d'un solveur QCSP performant, nous allons maintenant pouvoir étudier l'outil *coupure* (voir section 3) afin de valider son utilité.

5.2 L'impact de l'outil coupure dans QuaCode

Nous allons maintenant montrer l'intérêt de l'outil coupure lors de la résolution d'un QCSP. Nous comparons QuaCode avec et sans coupure pour tester son efficacité. Lorsque les informations sont disponibles, nous comparons également QuaCode avec Qecode, le solveur QCSP+. Pour ces expérimentations, nous allons utiliser la variante Fibonacci du jeu de Nim (voir exemple 1) fournie avec Qecode.

Lors de la comparaison de QuaCode avec et sans coupure, nous observons que le nombre d'échecs détectés pendant la recherche reste identique. En effet, l'outil coupure n'est utile que pour couper plus tôt les branches satisfiables de l'arbre de recherche, il n'améliore pas la détection des échecs.

L'outil coupure évite de parcourir des branches garanties d'être satisfiables de l'arbre de recherche. Le nombre de noeuds explorés ainsi que le nombre de propagations est donc moindre comme le montre la figure 10. Cette information n'étant pas disponible pour Qecode, nous comparons uniquement QuaCode avec et sans coupure. Nous observons qu'au delà de 9 allumettes, le nombre de propagations de QuaCode sans coupure est significativement plus grand que celui de QuaCode avec coupure. La figure 9 montre le nombre noeuds explorés pendant la recherche. Notons que ce nombre de noeuds ne correspond pas exactement au nombre de noeuds du véritable arbre de recherche en raison de la façon dont GeCode calcule cette information durant le backtrack. Nous avons légèrement modifié Qecode pour qu'il affiche cette information et ainsi l'inclure dans notre étude. Qecode et QuaCode avec coupure sont tous les deux plus rapides que QuaCode sans coupure. En effet, Qecode est conçu pour traiter plus efficacement les QCSP+ et éviter de parcourir

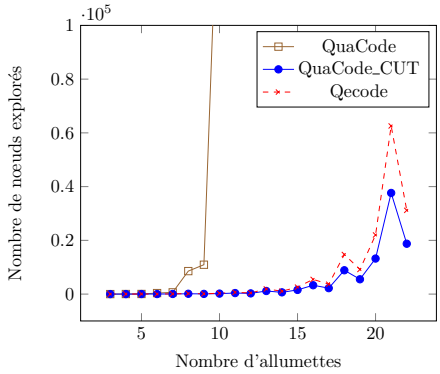


FIGURE 9 – Comparaison du nombre de noeuds explorés pour NimFibo

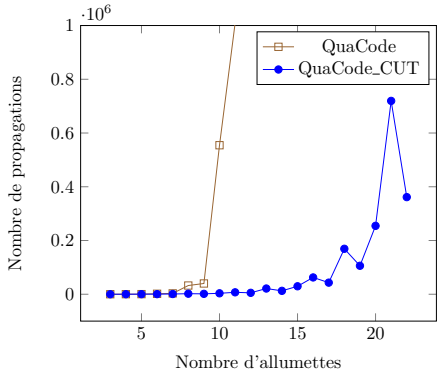


FIGURE 10 – Comparaison du nombre de propagations pour NimFibo

les branches évidentes conduisant obligatoirement à un succès. Les résultats de Qecode et QuaCode avec coupure suivent le même schéma. Cependant, QuaCode nécessite toujours moins de noeuds que Qecode. Sur la dernière instance, Qecode requiert deux fois plus de noeuds que QuaCode pour résoudre le problème.

L'utilisation de l'outil coupure peut conduire à un gain exponentiel du nombre de noeuds et de propagations. Cela se répercute immédiatement sur les temps de calculs. Dès que le nombre d'allumettes dépasse 15, le temps de résolution pour QuaCode sans coupure dépasse une heure alors qu'il reste sous une seconde pour QuaCode avec coupure (voir figure 11). Enfin, QuaCode est plus rapide que Qecode quelque soit l'instance NimFibo utilisée. Sur les dernières instances, il est deux fois plus rapide que Qecode. QuaCode et Qecode sont tous les deux basés sur GeCode, ils utilisent donc les mêmes propagateurs. Cependant, dans Qecode, la résolution des CSPs est laissée à GeCode et aucune connaissance provenant de la quantification des variables n'est utilisée pendant cette étape. QuaCode surveille les variables universelles (voir sec-

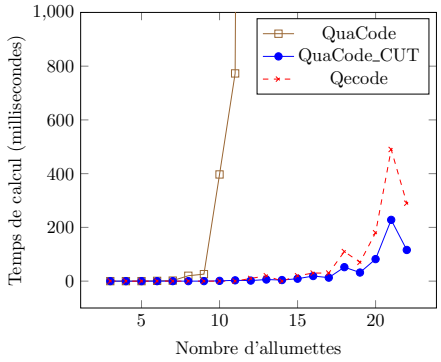


FIGURE 11 – Comparaison des temps de calculs pour NimFibo

tion 4) pour détecter au plus tôt certains échecs. Ces échecs ne sont pas décelables lorsque les quantificateurs sont existentiels, il est donc impossible à Qecode de les trouver lors de la résolution des sous-problèmes.

Le cadre QCSP+ est utile pour modéliser un jeu à deux joueurs à horizon fini. Nous avons montré qu'utiliser un solveur dédié aux QCSP+ n'est pas la seule façon de résoudre efficacement ces problèmes. En effet, nos expérimentations illustrent bien l'impact de l'outil coupure. Un de ses intérêts est d'adapter facilement un solveur QCSP afin de résoudre efficacement des problèmes QCSP+.

6 Conclusion

Dans cet article, nous avons présenté l'outil *coupure*, un outil utilisé pour éviter de parcourir des branches de l'arbre de recherche où le succès est garanti. Cet outil est sous la responsabilité de l'utilisateur qui doit utiliser sa connaissance du problème pour ajouter les coupures lors de la modélisation. Nous avons montré qu'ajouter cette connaissance dans QuaCode un solveur QCSP construit au dessus de GeCode améliore significativement la recherche d'une stratégie gagnante en évitant de parcourir certaines branches de l'arbre de recherche. Pour le cas des QCSP+, certaines coupures peuvent être ajoutées automatiquement car elles ne dépendent que des opérateurs liés aux tours de jeux du $\forall$-joueur.

Nous avons prouvé qu'une coupure dans le problème primal correspond à une branche échouée du dual. Ainsi, la connaissance sur le dual d'un problème peut-être injectée sous forme de coupures dans le primal. Résoudre le dual d'un problème est aussi difficile que de résoudre le primal. Ainsi, même si les coupures pouvaient être calculées automatiquement par analyse de la résolution du primal, le temps passé ne serait pas rentabilisé. En conséquence, fournir l'outil de coupure permettant à l'utilisateur d'ajouter lui même ces informations peut s'avérer être très utile.

En prolongement de ce travail, nous prévoyons d'étudier les *goods* trouvés pendant la résolution afin de générer automatiquement des coupures. L'apprentissage des *nogoods* a montré son efficacité pour résoudre des CSPs, nous pensons que l'apprentissage des *goods* pourrait de manière symétrique améliorer la recherche d'une stratégie gagnante dans un QCSP. Nous sommes également intéressés par d'autres façons de générer des coupes à faible coût comme par exemple en analysant la formule du problème initial à la découverte de disjonctions.

Références

[1] M. Benedetti, A. Lallouet, and J. Vautard. Modeling adversary scheduling with QCSP+. In *Proceedings of the 23th ACM Symposium on Applied Computing (SAC'08)*, pages 151–155, 2008.

[2] 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.

[3] M. Cadoli, A. Giovanardi, and M. Schaerf. An Algorithm to Evaluate Quantified Boolean Formulae. In *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI'98)*, pages 262–267, 1998.

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

[5] C. Jefferson, N. Moore, P. Nightingale, and K. Petrie. Implementing logical connectives in constraint programming. *Artificial Intelligence*, 174(16-17) :1407–1429, 2010.

[6] I. Stéphan. Un panorama sur les procédures de décision séquentielles pour le problème de validité des formules booléennes quantifiées. *Revue d'Intelligence Artificielle*, 26-1&2 :163–196, 2012.

[7] E. Tsang. Foundations of constraint satisfaction. *Academic Press, London*, 1993.

[8] G. Verger and C. Bessiere. BlockSolve : a Bottom-Up Approach for Solving Quantified CSPs. In *Proceedings of the 12th International Conference on Principles and Practice of Constraint Programming (CP'06)*, pages 635–649, 2006.

Un Solveur Complet pour les Constraint Games

Thi-Van-Anh Nguyen **Arnaud Lallouet**
GREYC, Normandie Université, Caen, France
{thi-van-anh.nguyen | arnaud.lallouet}@unicaen.fr

Résumé

La Théorie des Jeux étudie des situations dans lesquelles de multiples agents ayant des objectifs en conflits recherchent une décision collectivement satisfaisante. La question d'une représentation compacte des utilités des agents est d'une importance cruciale car la représentation classique d'un jeu à n joueurs est la donnée d'une matrice à n dimensions pour chaque joueur, ce qui est de taille exponentielle. Dans cet article, nous étudions le cadre des Jeux avec Contraintes, ou Constraint Games, dans lesquels les utilités sont représentées par des CSP. La Programmation par Contraintes permet de donner un modèle compact et élégant à de nombreux jeux. Nous distinguons deux types de Constraint Games : les jeux de satisfaction et les jeux d'optimisation. De plus, en ajoutant des contraintes dures, il est possible d'interdire certaines situations globales. Dans cet article, nous étudions une technique de recherche complète et nous montrons que notre solveur utilisant la représentation compacte des Constraint Games se compare favorablement au solveur classique Gambit.

Introduction

La Théorie des Jeux connaît un immense succès pour la modélisation de l'interaction entre multiples agents [21, 20]. Dans un jeu stratégique, chaque joueur doit choisir une action parmi un ensemble d'actions possibles et reçoit une récompense dont la valeur dépend aussi des actions choisies par les autres joueurs. Un des concepts de solution les plus connus est celui de l'*équilibre de Nash en stratégies pures* (PNE) dans lequel aucun joueur n'est en mesure d'améliorer son utilité en ne changeant que ses propres actions. Il y a de nombreux concepts de solution [24] mais les PNE ont l'avantage de préciser un choix de façon déterministe pour les joueurs. En effet, les PNE pour les jeux sont l'analogue des solutions dans les Constraint Satisfaction Problems (CSP) : certains jeux n'ont pas de PNE, certains PNE peuvent être préférés à d'autres. Ils sont néanmoins un outil fondamental pour l'étude des jeux.

La représentation de base d'un jeu est donnée par une multimatrice appelée *forme normale* dont la taille est exponentielle dans le nombre de joueurs. La taille intraitable de cette représentation est une sévère limitation pour une utilisation large de modèles basés sur les jeux. Ceci a motivé plusieurs travaux sur les représentations compactes. Certaines sont basées sur une limitation des interactions entre joueurs, comme les jeux graphiques [18] ou les jeux graphe-action [17] tandis que d'autres sont basées sur des restrictions de langage comme les Boolean Games [14, 9, 3] ou les Constraint Games [22].

Ici nous étudions les Constraint Games dans lesquels les utilités sont représentées par des CSP ou des Constraint Optimization Problems (COP). Les Constraint Games possèdent un langage de modélisation très riche permettant de représenter les buts des joueurs. En particulier, ils permettent de représenter de façon compacte la plupart des jeux classiques tels que les jeux de congestion [26], les jeux de réseaux [5], les problèmes d'ordonnancement stratégiques [28], et bien d'autres. De plus, il est facile d'exprimer des contraintes dures partagées par tous les joueurs [25] permettant de limiter l'ensemble des stratégies conjointes possibles et ainsi d'exclure des équilibres irréalistes.

En dépit du grand intérêt consistant à modéliser l'interaction stratégique multi-agents, il est très difficile de trouver un solveur retournant les PNE. A notre connaissance, il n'y a que le solveur Gambit [19], qui est considéré comme l'état de l'art. Toutefois, des transformations logiques non-compactes ont été étudiées [8, 12], ainsi que les ASP [7]. Pour les Boolean Games, une technique a été proposée pour les jeux à graphe d'interaction acyclique [2] ou en utilisant des techniques de marchandage [10]. Pour les Constraint Games, seul un solveur basé sur la recherche locale a été proposé [22]. Sans surprise, il permet de résoudre

de très grands jeux mais au prix de l'incomplétude. Cette situation est due en grande partie de la complexité de la recherche d'un PNE [13, 3].
Peu d'autres travaux mélangent jeux et Programmation par Contraintes. Dans [4], il est proposé de trouver un équilibre de Nash en stratégies mixte en utilisant les contraintes continues. D'autres cadres ont pour but de résoudre un problème combinatoire à l'aide de plusieurs agents, soit en attribuant les variables aux joueurs comme dans les DCOP [11] ou en laissant les agents choisir leurs variables à tour de rôle comme dans Game-SAT [29] ou Adversarial CSP [6].
Dans cet article, nous prouvons que les Constraint Games sont Σ_2^p-complets comme les Boolean Games et nous nous intéressons au problème de la recherche exhaustive de tous les PNE d'un jeu donné. Bien que la recherche d'un seul équilibre soit déjà un problème intéressant en soi, les trouver tous est une base qui permettra d'aller vers la recherche d'équilibres satisfaisant des conditions particulières. Par exemple, la correction de la recherche des équilibres de Nash Pareto-efficaces est directement conséquence de la complétude de la recherche des PNE. Nous avons implanté notre algorithme dans le solveur ConGa. Ce solveur est basé sur un calcul accéléré des équilibres basé sur une condition que nous appelons *consistance de Nash* et une détection incomplète mais paresseuse de certains états toujours dominés (Never Best Responses, ou NBR). Nous démontrons l'efficacité de notre approche sur des jeux de la suite Gamut [23] ainsi que sur des applications réalistes.

Constraint Games

Soit V un ensemble de variables et $D = (D_x)_{x\in V}$ la famille de leurs domaines (finis). Pour $W \subseteq V$, nous notons D^W l'ensemble des n-uplets sur W, soit $\Pi_{x\in W} D_x$. La projection d'un n-uplet (ou d'un ensemble de n-uplets) sur une variable (ou un ensemble de variables) est notée $|$: pour $t \in D^V$, $t|_x = t_x$, $t|_W = (t_x)_{x\in W}$ et pour $E \subseteq D^V$, $E|_W = \{t|_W \mid t \in E\}$. Pour $W, U \subseteq V$, la jointure de $A \subseteq D^W$ et $B \subseteq D^U$ est $A \bowtie B = \{t \in D^{W\cup U} \mid t|_W \in A \ \wedge \ t|_U \in B\}$. Quand $W \cap U = \emptyset$, nous notons la jointure des n-uplets $t \in D^W$ et $u \in D^U$ par (t, u). Une *constraint* $c = (W, T)$ est un couple composé d'un sous-ensemble $W = var(c) \subseteq V$ de variables et une relation $T = sol(c) \subseteq D^W$ (appelée *solutions*). Un Constraint Satisfaction Problems (CSP) est un ensemble de contraintes. Nous notons $sol(C) = \bowtie_{c\in C} sol(c)$ l'ensemble de ses solutions. Afin de simplifier l'exposition, nous identifions $sol(C)$ avec son extension cylindrique à toutes les variables de V (c.a.d avec n'importe quelle combinaisons de valeurs pour les variables qui n'apparaissent pas dans C).
Soit $\mathcal{P}$ un ensemble de n joueurs et V un ensemble fini de variables. L'ensemble des variables est partitionné entre les variables *contrôlées* $V_c = \bigcup_{i\in\mathcal{P}} V_i$ où V_i est le sous-ensemble de variables contrôlées par le joueur i, et V_E est l'ensemble des variables *non-contrôlées* ou *existentielles* ($V_E = V \setminus V_c$).

Definition 1 (Constraint Satisfaction Game) *Un* Constraint Satisfaction Game *(ou CSG) est un 4-uplet $(\mathcal{P}, V, D, G)$ où $\mathcal{P}$ est un ensemble fini de joueurs, V est un ensemble fini de variables composé d'une famille d'ensembles disjoints (V_i) pour chaque joueur $i \in \mathcal{P}$ et un ensemble V_E de variables* existentielles *disjoint des variables des joueurs, $D = (D_X)_{X\in V}$ est la famille de leurs domaines et $G = (G_i)_{i\in\mathcal{P}}$ est une famille de CSP sur V.*

Le CSP G_i est appelé le *but* du joueur i. Dans un CSG, tandis que le joueur i ne contrôle que ses variables V_i, sa satisfaction dépend aussi des variables contrôlées par les autres joueurs. Dans la plupart des cas, les variables existentielles sont utilisées pour des calculs et sont dépendantes fonctionnellement des variables de décision, mais ceci n'a bien sûr pas de caractère obligatoire.
Une *stratégie* pour le joueur i est une affectation de ses variables contrôlées V_i. Un *profil de stratégies* $s = (s_i)_{i\in\mathcal{P}}$ est la donnée d'une stratégie pour chaque joueur.

Definition 2 (Stratégie gagnante) *Un profil de stratégies s est gagnant pour i s'il satisfait le but de i : $s \in sol(G_i)$.*

Un CSG peut être interprété comme un jeu au sens classique avec une utilité booléenne dont la valeur vaut 1 quand le but du joueur est satisfait et 0 sinon.
Nous notons s_{-i} la projection de s sur $V_{-i} = V \setminus V_i$. Pour un profil de stratégies s, un joueur i a une *déviation bénéfique* si $s \notin sol(G_i)$ et $\exists s'_i \in D^{V_i}$ tel que $(s'_i, s_{-i}) \in sol(G_i)$. Une déviation bénéfique correspond au fait qu'un joueur va tenter de modifier son affectation des variables qu'il contrôle s'il pense pouvoir obtenir une meilleure utilité. Un n-uplet s est une *meilleure réponse* pour le joueur i si ce joueur n'est pas en mesure de trouver une déviation bénéfique. Nous pouvons maintenant définir la notion de solution d'un CSG par un équilibre de Nash en stratégies pures (PNE) :

Definition 3 (PNE) *Un profil de stratégies s est un* Equilibre de Nash en Stratégies Pures (ou PNE) *d'un CSG $\mathcal{C}$ si et seulement si aucun joueur n'a de déviation bénéfique. Ou, ce qui revient au même, si s est une meilleure réponse pour tous les joueurs.*

Theorem 1 *Les CSG sont Σ_2^p-complets.*

Proof 1 *La preuve est similaire à la preuve de Σ_2^p-complétude des jeux booléens [3].*

Les buts des joueurs peuvent être considérés comme des contraintes molles ou des préférences car il est possible que le but d'un joueur ne soit pas satisfait dans un PNE si ce joueur ne peut de toutes façon pas le satisfaire. Au pire tous les joueurs peuvent être dans ce cas. Mais il peut arriver que certaines règles interdisent certaines situations de se produire. Il est donc naturel de rejeter ces profils de stratégies en posant des *contraintes dures* partagées par l'ensemble des joueurs [25]. Il est facile de représenter ces contraintes dans les CSG en ajoutant simplement un CSP global au problème :

Definition 4 (CSG avec Contraintes Dures) *Un* Constraint Satisfaction Game avec Contraintes Dures *(ou CSG-HC) est un 5-uplet $(\mathcal{P}, V, D, C, G)$ où $(\mathcal{P}, V, D, G)$ est un CSG et C est un CSP sur V.*

Il est donc utile de distinguer un profil de stratégies qui ne satisfait le but d'aucun joueur d'un profil qui ne satisfait pas les contraintes dures. Le premier peut être un PNE tandis que le second non. Les contraintes dures améliorent l'expressivité des CSG (sans en changer la complexité).
En ajoutant un critère d'optimisation, il est possible de représenter les jeux classiques. Un *Constraint Optimization Game* (ou COG) est une extension des CSG dans laquelle chaque joueur tente d'optimiser un objectif. Pour cela, on donne un critère d'optimisation pour chaque joueur i de la forme $\min(x)$ ou $\max(x)$ où $x \in V$ est une variable devant être optimisée par le joueur i.

Definition 5 (Constraint Optimization Game) *Un* Constraint Optimization Game *(ou COG) est un 5-uplet $(\mathcal{P}, V, D, G, opt)$ où $(\mathcal{P}, V, D, G)$ est un CSG et $opt = (opt_i)_{i\in\mathcal{P}}$ est une famille de critères d'optimisation pour chaque joueur de la forme* $\min(x)$ *ou* $\max(x)$ *où $x \in V$.*

Une stratégie gagnante pour le joueur i est toujours un profil qui satisfait G_i. Toutefois, la notion de déviation bénéfique doit être adaptée. On note $<_{opt_i}$ l'ordre (partiel) sur les profils de stratégies tel que $s <_{opt_i} s'$ si $s_{-i} = s'_{-i}$ et $s|_x < s'|_x$ quand $opt_i = \min(x)$ (resp. $s|_x > s'|_x$ quand $opt_i = \max(x)$). Pour un profil s, un joueur i a une *déviation bénéfique* si $\exists s'_i \in D^{V_i}$ tel que $s' = (s'_i, s_{-i}) \in sol(G_i)$ et $s' <_{opt_i} s$. Avec cette adaptation, la notion de solution est la même que pour les CSG. De plus, les COG peuvent être étendus avec des contraintes dures de la même façon que les CSG, ce qui donne les COG-HC.

Exemples de Constraint Games

Dans cette section, nous montrons que des jeux complexes peuvent être facilement exprimés avec les Constraint Games.

Example 1 (Location Game) *Dans cet exemple inspiré par [15], n vendeurs de glaces $\mathcal{P} = \{1, 2, .., n\}$ désirent choisir un emplacement numéroté de 1 à m pour installer leur stand dans une rue. Chaque vendeur i a fixé le prix d'une glace à p_i et on suppose qu'il y a un client à chaque emplacement. Il est impossible pour deux vendeurs de choisir le même emplacement. Les clients choisissent leur vendeur en minimisant la somme de la distance entre leur maison et le vendeur le plus proche et le prix de la glace.*
Nous avons besoin des variables existentielles suivantes (dont la valeur est déterminée fonctionnellement par l'emplacement l_i choisi par le vendeur i) :
- *$cost_{ic}$: définit le coût que le client c doit payer s'il choisit le vendeur i.*
- *min_c : définit le coût minimal que le client c doit payer pour sa glace.*
- *$choice_{ic}$: variable booléenne qui vaut 1 si le client c choisit le vendeur i.*
- *$benefit_i$: définit le bénéfice du vendeur i.*

The Location Game (LG) can be easily modelled by a COG-HC in which each seller wants to maximize her profit :
- $\mathcal{P} = \{1, \ldots, n\}$
- $\forall i \in \mathcal{P}, V_i = \{l_i\}$
- $\forall i \in \mathcal{P}, D(l_i) = \{1, \ldots, m\}$
- *les contraintes dures C sont les suivantes :*
  - *les vendeurs occupent des emplacements différents :* all_different$(l_1, l_2, \ldots, l_n)$
  - $\forall i \in \mathcal{P}, \forall c \in [1..m]$, $cost_{ic} = |c - l_i| + p_i$
  - $\forall c \in [1..m]$, $min_c = \min(cost_{1c}, \ldots, cost_{nc})$
  - $\forall c \in [1..m], (min_c = cost_{ic}) \leftarrow (choice_{ic} = 1)$. *On utilise une implication et non une équivalence car il peut arriver qu'un client obtienne le même prix de deux vendeurs. La contrainte de somme suivante assurera qu'un client n'ira que chez un seul vendeur.*
  - $\forall c \in [1..m], \sum_{i\in\mathcal{P}} choice_{ic} = 1$
- $\forall i \in \mathcal{P}$, *$G_i$ contient la contrainte suivante :* $benefit_i = p_i . \sum_{c\in[1..m]} choice_{ic}$
- $\forall i \in \mathcal{P}$, *le critère d'optimisation est* $Opt_i = \max(benefit_i)$

Une caractéristique intéressante de cet exemple est qu'il utilise une contrainte globale (ici all_different*). Cela montre l'intérêt des contraintes dures dans la modélisation de jeux réalistes. Il est possible de transformer ce problème en un CSG en fixant un profit minimal mp_i pour chaque joueur i et en ajoutant la contrainte que le joueur i est satisfait si son bénéfice*

dépasse mp_i : on ajoute $_i \geq mp_i$ to G_i à la place de la condition d'optimisation. Dans la version de ce jeu définie par Gamut [23], les vendeurs choisissent le prix des glaces et non leur emplacement car il n'y a aucun moyen d'exclure le fait que deux joueurs peuvent se mettre au même endroit.

Example 2 (Cloud Resource Allocation Game) *L'allocation de ressources est un élément central dans la gestion du calcul dans le nuage où les clients utilisent et payent des ressources de calcul à la demande. Afin de gérer les conflits d'intérêts entre différents clients, [16] ont proposé le cadre du* CRAG *(Cloud Resource Allocation Game) dans lequel l'allocation de resosources est définie par un équilibre de Nash.*

Un fournisseur d'informatique dans le nuage possède un ensemble $\mathcal{M} = \{M_1, \ldots, M_m\}$ de m machines, chaque machine M_j ayant une capacité c_j représentant la quantité de ressource disponible (par exemple heure-CPU, mémoire). Le coût d'utilisation d'une machine j est donné par $l_j(x) = x \times u_j$ où x est le nombre de ressources demandées et u_j un coût unitaire. Un ensemble de n clients $\mathcal{P} = \{1, 2, .., n\}$ veut utiliser simultanément le nuage. Le client $i \in \mathcal{P}$ a m_i tâches $\{T_{i1}, ..., T_{im_i}\}$ à exécuter, de capacité respective $\{d_{i1}, ..., d_{im_i}\}$. Chaque client $i \in \mathcal{P}$ choisit de manière égoïste une allocation r_{ik} pour sa tâche T_{ik} $(k \in 1..m_i)$ et souhaite minimiser son coût $cost_i = \sum_{k=1..m_i} l_{r_{ik}}(d_{ik})$. Nous supposons que le fournisseur possède suffisamment de ressources pour satisfaire la demande de tous les clients : $\sum_{i\in[1..n]} \sum_{k\in[1..m_i]} d_{ik} \leq \sum_{j\in[1,..m]} c_j$. Ce problème est modélisé par le COG-HC suivant :

- $\mathcal{P} = \{1, .., n\}$
- $\forall i \in \mathcal{P}, V_i = \{r_{i1}, ..., r_{im_i}\}$
- $\forall i \in \mathcal{P}, \forall k \in [1, ..., m_i], D(r_{ik}) = \{1, \ldots, m\}$
- *C est composé des contraintes dures suivantes :*
  - *contraintes de channelling pour les variables booléennes établissant que la machine j est demandée par la tâche t_{ik} : $(r_{ik} = j) \leftrightarrow (choice_{ijk} = 1)$*
  - *contraintes de capacité : $\forall j \in [1, .., m]$, $\sum_{i\in[1..n]} \sum_{k\in[1..m_i]} choice_{ijk} \times d_{ik} \leq c_j$*
- *$\forall i \in \mathcal{P}, G_i$ est composé de :*

$$cost_i = \sum_{j=1..m} \sum_{k=1..m_i} choice_{ijk} \times l_j(d_{ik})$$

- $\forall i \in \mathcal{P}, Opt_i = Minimize\ (cost_i)$

Parmi les autres exemples intéressants, on trouve les jeux de réseaux [5], l'ordonnacement stratégique [28], ou les jeux de la suite Gamut [23].

Techniques de coupe

Un algorithme naturel pour trouver les PNE consiste à générer et tester. Curieusement, cet algorithme est le seul connu pour trouver un PNE [27] et constitue la base de l'algorithme du solveur de référence Gambit [19]. D'après le résultat de complexité (théorème 1), il est toutefois improbable qu'un algorithme efficace (polynomial) existe. Dans la suite, afin de simplifier la présentation, nous supposons que chaque joueur i ne contrôle qu'une seule variable x_i de domaine D_i. L'extension à plusieurs variables de décision par joueur n'est pas difficile et notre solveur ConGa n'a d'ailleurs pas cette limitation, de nombreux exemples ne pouvant se contenter d'une variable par joueur.

Algorithm 1 enum1

```
1: function ENUM1(Game CG) : setof tuples
2:     Nash ← ∅
3:     for s ∈ D^{V_c} do
4:         if IsNash(s) then
5:             Nash = Nash ∪ {s}
6:         end if
7:     end for
8:     return Nash
9: end function

10: function ISNASH(tuple s) : boolean
11:     for i ∈ P do
12:         for v ∈ D_i, v ≠ s_i do
13:             if (s_{-i}, v) <_{opt_i} s then
14:                 return false
15:             end if
16:         end for
17:     end for
18:     return true
19: end function
```

L'algorithme *enum1* (Algorithme 1) consiste à énumérer tous les profils de stratégies et les tester pour vérifier qu'aucun joueur n'a de déviation bénéfique. On passe au profil suivant dès qu'une telle déviation est détectée. L'exemple suivant montre que certaines déviations sont testées plusieurs fois.

Example 3 *Soit G le jeu à deux joueurs défini par la bimatrice suivante :*

		y		
		1	*2*	*3*
x	*a*	$(0,1)_\alpha$	(1,0)	(1,0)
	b	$(0,1)_\beta$	(0,0)	(1,0)
	c	(1,0)	(1,1)	(0,0)

Supposons que l'énumération commence par le joueur x. Le premier n-uplet énuméré est $(a, 1)$ indiqué par α. La déviation est testée pour le joueur y qui reste

stable. La déviation est ensuite testée pour le joueur x et un mouvement vers $(c, 1)$ est trouvé. Ce n-uplet n'est donc pas un PNE. Le candidat suivant est $(b, 1)$ indiqué par β. Ce n-uplet est testé pour y et encore une fois on ne trouve pas de déviation. Mais lorsqu'on teste le joueur x, on retrouve la même déviation vers $(c, 1)$ qui avait déjà été calculée pour x en α.

Cette forme d'enlisement est une bonne motivation pour étudier les techniques de recherche et de coupe pour les Constraint Games. Afin de présenter notre technique, nous commençons par rappeler [13] où les auteurs introduisent (à l'origine pour les jeux graphiques) un CSP composé de *contraintes de Nash* pour représenter les meilleures réponses de chacun des joueurs.

Definition 6 (GGS-CSP) *Soit $CG = (\mathcal{P}, V, D, G, opt)$ un COG. La* contrainte de Nash *du joueur $i \in \mathcal{P}$ est $g_i = (V_c, T)$ où $T = \{t \in D^V \mid \nexists t' \in D^V \text{ t.q. } t' <_{opt_i} t\}$. Le* GGS-CSP $\mathcal{G}(CG)$ *de CG est défini par l'ensemble des contraintes de Nash pour chaque joueur.*

Ce CSP a comme propriété importante que ses solutions sont les PNE du jeu en question :

Theorem 2 *([13]) t est un PNE de $CG \leftrightarrow t \in sol(\mathcal{G}(CG))$*

Il en découle qu'un PNE d'un Constraint Game CG a un support dans chacune de ses contraintes de Nash.
Notre technique consiste à réaliser un parcours de l'espace de recherche en affectant les variables de chaque joueur à tour de rôle selon un ordre prédéfini sur $\mathcal{P}$. Pour chaque profil candidat, nous recherchons son support avec un calcul incrémental des contraintes de Nash. Chaque déviation trouvée est enregistrée dans une table pour chacun des joueurs. On évite enseuite de recalculer les déviations déjà enregistrées grâce à une recherche dans les tables.
Toutefois, comme nous étudions ici des jeux quelconques sans faire d'hypothèse sur leur structure, chaque contrainte de Nash a comme arité la totalité des variables contrôlées, ce qui peut poser des problèmes d'encombrement en mémoire. Toutefois, on peut déjà noter qu'un n-uplet non stocké ou supprimé ne pose pas de problème de correction. Il se peut simplement qu'une déviation soit calculée plusieurs fois. On peut donc en dernier recours limiter la taille des tables. Mais en pratique deux choses limitent l'extension des tables.
Premièrement, les tests de déviation sont faits dans l'ordre inverse de l'énumération. Ce qui veut dire qu'un n-uplet testé pour le premier joueur devra être meilleure réponse pour tous les autres joueurs. Dans la plupart des problèmes que nous avons étudié, cela limite le nombre de n-uplets atteignant les niveaux élevés.
Deuxièmement, il est possible de supprimer un n-uplet t d'une table quand on est sûr qu'aucun n-uplet t' ne déviera plus vers t. Cette propriété est donnée par le théorème suivant d'indépendance des sous-jeux. Soit un Constraint Game $CG = (\mathcal{P}, V, D, G, opt)$. Un jeu $CG' = (\mathcal{P}, V, D', G, opt)$ est un sous-jeu de CG si $\exists i \in \mathcal{P}, D'_i \subseteq D_i$ et $i \neq j \rightarrow D'_j = D_j$. On note par $br_i(t) = \{t' \in sol(G_i) \mid t'_{-i} = t_{-i}\}$ l'ensemble des stratégies meilleures réponses à partir de t pour le joueur i.

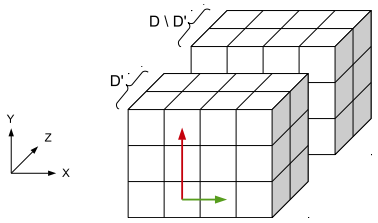


FIGURE 1 – Indépendance des sous-jeux

Proposition 3 *Soit CG un constraint game et CG' un sous-jeu de CG tel que $D'_i \subseteq D_i$. Soit $D'' = D \setminus D'$, $t' \in D'^V$ et $t'' \in D''^V$. Alors $\forall j \in \mathcal{P}, j \neq i \rightarrow br_j(t) \cap br_j(t') = \emptyset$.*

Proof 3 *Par l'absurde. Supposons qu'il existe $t' \in D'^V$ et $t'' \in D''^V$ tel que $br_j(t') \cap br_j(t'') \neq \emptyset$. Soit $s \in br_j(t') \cap br_j(t'')$. Alors puisque $s \in br_j(t')$, $s_i \in D'^{V_i}$. Puisque $s \in br_j(t'')$, $s_i \in D''^{V_i}$. D'où la contradiction.* □

Cette proposition est illustrée en Figure 1 : si l'on coupe l'espace de recherche selon le joueur Z, les meilleures réponses des joueurs X et Y sont forcées de rester dans leur sous-espace respectif par rapport à Z.
En appliquant inductivement la Proposition 3 sur une séquence d'affectations de stratégies pour les joueurs de 1 à k avec $k < n$, on voit que deux branches de l'arbre de recherche ne partageront pas de meilleures réponses pour les joueurs suivants. On peut donc réinitialiser les tables des joueurs en aval une fois une branche explorée.
La dernière optimisation consiste en l'élimination de stratégies qui ne sont jamais meilleures réponses (*never best responses*, ou NBR).

Definition 7 *Une stratégie s_i pour le joueur i est une* never best response *si $\forall t_{-i}, \exists s'_i$ tel que $(s'_i, t_{-i}) <_{opt_i} (s_i, t_{-i})$.*

L'élimination itérative des NBR est une technique de coupe préservant la correction pour les jeux [1]. On peut noter qu'elle est plus forte que l'élimination des

stratégies fortement dominées. Mais hélas sa détection est très coûteuse dans le cas d'un jeu à n joueurs car il est nécessaire de s'assurer que l'action ne sera jamais choisie pour tous les profils des autres joueurs. Toutefois, être une NBR dans un sous-jeux est une condition suffisante pour ne pas être un équilibre :

Proposition 4 *Soit CG un constraint game et CG' un sous-jeux de CG tel que $D'_i \subseteq D_i$. Soit $s_j \in D'_j$ une NBR de CG' avec $j \neq i$. Alors, pour tout s_{-j}, si $s = (s_j, s_{-j})$ est un PNE, alors $s_i \notin D'_i$.*

Proof 4 *Par l'absurde. Supposons qu'il existe un PNE $s = (s_j, s_{-j})$ avec $(s_{-j})_i \in D'_i$. Puisque s est un PNE, on a $\forall k \in \mathcal{P}, s_k \in br(s)|_k$. Et puisque s_j est une NBR pour j dans CG', il existe s'_j tel que $(s'_j, s_{-j}) <_{opt_j} (s_j, s_{-j})$. Donc $s_j \notin br(s)|_j$.* □

En appliquant inductivement la proposition 4 sur une séquence d'affectation de stratégies pour les joueurs de 1 à k avec $k < n$, on voit que si l'on détecte qu'une valeur v est NBR pour le joueur k dans le sous-jeux défini par la suite d'affectations, alors cette valeur ne pourra pas participer à un PNE et nous pouvons la supprimer. Elle ne sera toutefois supprimée que dans la branche courante et pourra réapparaître par la suite avec un préfixe d'affectations différent.

Un algorithme d'énumération des PNE

Nous proposons un algorithme de recherche arborescente pour le calcul de tous les PNE. Il est basé sur les trois idées suivantes :

Algorithm 2 ConGa

```
1: global :
2:     BR : array[1..n] of tuples
3:     cnt : array[1..n] of integer
4:     Nash : set of tuples
5:     S : global solver

6: function ConGa(Game CG) : setof tuples
7:     Nash ← ∅
8:     Initialize solver S with hard constraints
9:     A ← D
10:    enum(A, 1)
11:    return Nash
12: end function
```

– tous les candidats (sauf ceux détectés comme NBR) sont générés dans un ordre lexicographique ;
– les solutions non-dominées de chaque joueur sont enregistrées dans une table ;
– quand un test de dominance est exécuté, on recherche d'abord une déviation dans la table des meilleures réponses calculées.

Nous supposons donné un ordre sur les joueurs de 1 à n. La fonction principale (Algorithm 2) lance la traversée récursive (Algorithm 3) en commençant par le joueur 1. Nous distinguons lors du calcul les domaines initiaux des variables (appelés D et utilisés pour calculer les déviations) des domaines courants réduits lors de l'exploration de l'arbre de recherche (appelés A et réduits par l'arc-consistance sur les contraintes dures).

Algorithm 3 enum

```
1: procedure enum(domains A, int i)
2:     status ← S.propagate(A)
3:     if status then
4:         if i > n then
5:             checkNash(tuple(A),n)
6:         else
7:             BR[i] ← ∅
8:             cnt[i] ← Π_{j>i}|D_j|
9:             while A_i ≠ ∅ do
10:                choose v ∈ A_i
11:                B ← A
12:                enum((B_{-i}, (B_i = {v})), i + 1)
13:                A_i ← A_i − {v}
14:                if cnt[i] ≤ 0 then
15:                    checkEndOfTable(A, i)
16:                    break
17:                end if
18:            end while
19:        end if
20:    end if
21: end procedure
```

La propagation des contraintes dures permet d'assurer qu'aucun n-uplet interdit ne sera exploré. Si la propagation retourne *false*, alors au moins un des domaine a été vidé et cela signifie qu'il n'y a pas de solution dans ce sous-espace. Sinon, les domaines de A sont réduits par arc-consistance. Les valeurs des variables de chaque joueur sont énumérées récursivement. Quand un n-uplet complet est obtenu en fin de récursion, il est testé pour savoir s'il est un PNE (ligne 5) par l'algorithme 4. Si ce n'est pas le cas, il reste au moins un domaine à explorer. Chaque joueur i possède une table $BR[i]$ de meilleures réponses, initialement vide, et un compteur $cnt[i]$ initialisé à la taille du sous-espace engendré par les variables qui suivent dans l'ordre sur les joueurs. Dans un souci d'efficacité, les tables $BR[i]$ sont implantées par des arbres de recherche. car l'insertion et la suppression peuvent être réalisées en $O(|\mathcal{P}|)$. Après l'appel récursif de *enum*, on teste teste si tous le sous-espace situé en aval d'un joueur i a été testé pour déviation. Si c'est le cas, toutes les autres valeurs pour i sont des NBR et on peut remonter au noeud parent (lignes 14-17). Ce saut en arrière est effectué après une exploration des valeurs du sous-espace inexploré

stockées dans la table (fonction *checkEndOfTable*, Algorithme 6).

Algorithm 4 checkNash

```
1: procedure checkNash(tuple t, int i)
2:     if i = 0 then
3:         Nash ← Nash ∪ {t}
4:     else
5:         d ← search_table(t, BR, i)
6:         if d = ∅ then
7:             d ← deviation(t, i)
8:             if d = ∅ then
9:                 d ← D_i
10:            end if
11:            insert_table(i, BR, d)
12:            cnt[i] --
13:        end if
14:        if t_i ∈ d then
15:            checkNash(t, i − 1)
16:        end if
17:    end if
18: end procedure
```

La procédure *checkNash* dans l'algorithme 4 vérifie si un joueur peut faire une déviation bénéfique à partir d'un n-uplet. Puisque l'exploration est faite niveau par niveau, la vérification démarre par le niveau le plus profond. En premier, le n-uplet est recherché dans la table du joueur (ligne 5). S'il n'est pas trouvé, un solveur pour le but G_i est appelé dans la fonction *deviation* décrite dans l'algorithme 5 (ligne 7). Cette fonction retourne l'ensemble d des meilleures réponses pour le joueur i à partir du n-uplet t. Il peut y avoir plusieurs déviations. Dans un CSG, cela signifie simplement que le but a plusieurs solutions en changeant les variables contrôlées par le joueur i. Dans un COG, cela signifie que la valeur optimale est atteinte en plusieurs points. Si d est vide, alors il n'existe pas d'action pour i satisfaisant le but. Dans ce cas on retourne le domaine initial puisque ce joueur n'a pas de préférence particulière étant donné que son but est toujours insatisfait.

Algorithm 5 deviation

```
1: function deviation(tuple t, int i) : set of integer
2:     d ← ∅
3:     Initialize solver S_i with G_i (and opt_i for a COG)
4:     add constraints x_j = t_j for all j ≠ i
5:     sol ← S_i.getSolution()
6:     while sol ≠ nil do
7:         d ← d ∪ {sol}
8:         sol ← S_i.getSolution()
9:     end while
10:    return d
11: end function
```

Algorithm 6 checkEndOfTable

```
1: procedure checkEndOfTable(domain A, int i)
2:     for all t ∈ BR[i] such that t ∈ Π_{i=1..n} A_i do
3:         checkNash(t, n)
4:     end for
5: end procedure
```

La procédure *checkEndOfTable* décrite dans l'algorithme 6 est appelée quand le sous-espace a été exploré et juste avant d'effectuer le retour arrière. Elle vérifie que tous les n-uplets appartenant à la zone inexplorée sont bien testés comme candidats PNE. Un exemple de retour arrière est donné en Figure 2.

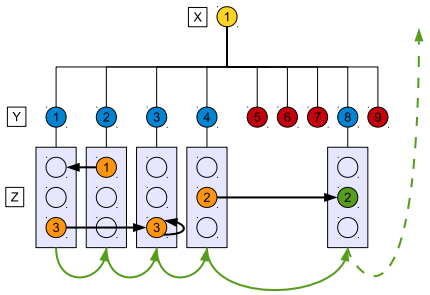


Figure 2 – Détection online des never best responses

Dans cet exemple, le domaine des joueurs Y et Z est respectivement de taille 9 et 3. Donc $cnt[Y]$ est initialisé à 3. Tous les n-uplets testés sont de la forme $1yz$ où 1 est la valeur sur X. Si nous supposons que les n-uplets 113, 121, 133 et 142 sont stable pour Z, ces n-uplets sont remontés au niveau de Y pour être testés. Les flèches pleines montrent les déviations trouvées et enregistrées pour Y. On voit dans cet exemple qu'en explorant seulement les valeurs 1, 2, 3 et 4 pour Y, on obtient un parcours complet du sous-espace défini par Z (toutes les valeurs du domaines de Zont été explorées). Donc après l'exploration de $Y = 4$, on en déduit que le sous-espace de Y ne peut plus contenir de PNE avec $X = 1$, sauf éventuellement pour la valeur 182 enregistrée dans la table de Y. Toutes les valeurs restantes de Y sont des NBR. Il est donc suffisant de tester ce dernier n-uplet grâce à *checkEndOfTable* et nous pouvons remonter au joueur X (lignes pointillées). En général, *checkEndOfTable* teste tous les n-uplets de $BR[Y]$ qui appartiennent à l'espace non encore exploré. Cette détection des NBR est incomplète mais en échange est très peu coûteuse puisqu'il n'y a qu'un compteur à maintenir. Notons également que par la proposition 3, quand on passe à $X = 2$, la table pour Y peut être réinitialisée car aucun n-uplet ne déviera plus vers un n-uplet où $X = 1$.

Proposition 5 *ConGa est correct et complet.*

Nom	Gen NF		Gambit	enum1			ConGa			#PNE
	Tps	***Taille***	***Tps***	***Tps***	***#Cand***	***#Dev***	***Tps***	***#Cand***	***#Dev***	
CG.7.15	253	5.1	*MO*	70	1.7E+8	1.8E+8	**27**	2.1E+7	1.3E+7	630
CG.8.15	4613	89	*MO*	1019	2.5E+9	2.7E+9	**371**	3.1E+8	1.9E+8	1680
CG.9.15	*TO*	–	–	17361	3.8E+10	4.2E+10	**5880**	4.9E+9	2.9E+9	5040
GTTA.3.100	1	0.1	17	4	1.0E+6	1.3E+6	**0**	1.0E+4	1.0E+4	1
GTTA.4.100	113	1.7	1844	312	1.0E+8	1.3E+8	**10**	1.0E+6	1.0E+6	1
GTTA.5.100	*TO*	205	*MO*	4032	1.0E+10	1.3E+10	**778**	1.0E+8	1.0E+8	1
LG(GV).2.1000	1	0.01	134	339	1.0E+6	1.0E+6	**6**	2.0E+3	1.5E+3	0
LG(GV).2.2000	6	0.04	655	1441	4.0E+6	4.0E+6	**31**	4.0E+3	3.5E+3	0
LG(GV).2.3500	17	0.1	5337	6789	1.2E+7	1.2E+7	**93**	7.0E+3	6.0E+3	0
LG(GV).2.5000	34	0.2	7786	20000	2.5E+7	2.5E+7	**201**	1.0E+4	9.0E+3	0
LG(GV).2.20000	552	3.7	*MO*	*TO*	–	–	**3578**	4.0E+4	3.9E+5	0
MEG.3.100	1	0.1	13	0	1.0E+6	1.0E+6	**0**	1.9E+4	1.5E+4	100
MEG.4.100	91	1.9	1555	28	1.0E+8	1.0E+8	**6**	1.9E+6	1.3E+6	100
MEG.5.100	*TO*	241	*MO*	2082	1.0E+10	1.0E+10	**403**	1.9E+8	1.2E+8	100
MEG.30.2	8784	91	*MO*	**423**	1.1E+9	2.1E+9	503	5.4E+8	1.1E+9	2
MEG.35.2	TO	–	–	**15933**	3.4E+10	6.9E+10	*TO*	–	–	2
TD.3.99	3	0.1	14	0	9.7E+5	9.8E+5	**0**	1.9E+4	1.5E+4	1
TD.4.99	76	1.9	1572	26	9.6E+7	9.7E+7	**7**	1.9E+6	1.3E+6	1
TD.5.99	8930	119	*MO*	2028	9.1E+9	9.6E+9	**446**	1.8E+8	1.2E+8	1
CRAG.7.9	N/A	N/A	N/A	323	4.7E+6	5.3E+6	**57**	1.0E+6	5.9E+5	1
CRAG.8.9	N/A	N/A	N/A	3300	4.2E+7	4.8E+7	**540**	9.5E+6	5.3E+6	1
CRAG.9.9	N/A	N/A	N/A	–	3.8E+8	4.3E+8	**5022**	4.3E+7	4.8E+7	1
LG(HC).4.30	N/A	N/A	N/A	26	6.5E+5	8.0E+5	**6**	1.4E+5	4.4E+4	24
LG(HC).5.30	N/A	N/A	N/A	778	1.7E+7	2.1E+7	**257**	4.1E+6	1.2E+5	240
LG(HC).6.30	N/A	N/A	N/A	*TO*	–	–	**13180**	1.1E+8	3.2E+7	2160

TABLE 1 – Results for Gamut and other games

Proof 5 *La correction vient de la correction du test des PNE (théorème 2). Un PNE découvert a forcément été testé en tant que meileure réponse pour chaque joueur. Soit le n-uplet a été trouvé dans une table en tant que déviation d'un autre n-uplet, soit il a directement été testé par le solveur sur le but du joueur. La complétude vient du parcours exhaustif de l'espace de recherche et de la correction de la coupe des NBR.* □

Experiments

Nous avons réalisé des expérimentations sur des jeux classiques de la suite Gamut [23] et sur des jeux avec contraintes dures. Les résultats sont résumés dans la table 1 dans laquelle le nom du jeu est suivi par le nombre de joueurs et la taille des domaines. Les jeux de Gamut sont CG (Congestion Game), GTTA (Guess Two Third Average), LG(GV) (Location Game, Gamut version), MEG (Minimum Effort Game) et TD (Traveller's Dilemma). Leur description peut être trouvée dans [23]. Les autres jeux sont LG(HC) (Location Game avec contraintes dures, exemple 1) et CRAG (Cloud Resource Allocation Game, exemple 2).

Pour chaque instance, nous avons comparé notre solveur ConGa au solveur Gambit et à un solveur que nous avons appelé *enum1* (Algorithme 1). Ce solveur travaille comme Gambit en examinant chaque n-uplet et la seule différence est qu'il travaille sur la représentation compacte des Constraint Games au lieu des multimatrices. Les expériences ont été réalisées sur un serveur de calcul AMD Opteron 6174 possédant 4 processeurs totalisant 48 coeurs à 2,2 GHz et avec 256 Go de RAM. Dans la table 1, les temps sont donnés en secondes, et la notation $aE{+}b$ désigne $a \times 10^b$.
Les expériences menées avec Gambit sont composées de deux parties : premièrement nous générons la forme normale du jeu (colonne Gen NF), puis nous lançons le solveur *gambit-enumpure* sur cette forme normale afin de trouver tous les PNE. Nous avons mesuré le temps nécessaire à sa génération (avec un time-out de 9000 secondes), sa taille (en Go), puis le temps de résolution par Gambit (avec un time-out de 20 000 secondes). TO signifie *Time Out*, MO *Memory Out* et "–" signifie que l'information n'est pas pertinente (par exemple, si le temps de génération dépasse le time-out, il n'est pas possible de lancer la résolution). Comme on pouvait s'y attendre, la taille de la forme normale devient très vite intraitable et excède la capacité mémoire de Gambit,

bien que nous ayons constaté qu'il puisse gérer des matrices d'environ 2 Go.
Pour *enum1* et *ConGa*, nous avons mesuré le temps nécessaire à la résolution d'une instance (avec aussi un time-out de 20 000 secondes), le nombre de profils candidats et le nombre de déviations testées. Avec un raisonnement simple, on peut trouver que le nombre de candidats examinés par *enum1* est simplement $|D^{V_c}|$, et le nombre de tests de déviations est compris entre le nombre de candidats (s'il n'y a aucun PNE et que la détection est immédiate) et une borne supérieure de $|D^{V_c}| \times |\mathcal{P}|$ (si tous les candidats sont des PNE). Sans surprise, nous voyons que *ConGa* coupe la plupart du temps une bonne partie de l'espace de recherche, grâce à la détection des NBR. Mais de façon plus intéressante et la plupart du temps, il évite aussi des tests de déviation, ce qui signifie que la solution est trouvée dans une table avant que le test soit lancé. Un contre-exemple est le Minimum Effort Game avec une taille de domaine de 2 (MEG.30.2 et MEG.35.2) pour lequel ni les tables ni la détection de NBR ne fonctionnent car les domaines sont trop petits. On pourra aussi noter que les jeux avec contraintes dures ne sont pas exprimables en forme normale et donc ne peuvent pas être résolus par Gambit. Ceci est indiqué par N/A (*non applicable*). Dans toutes les autres expériences, ConGa surpasse Gambit et *enum1* par au moins un ordre de magnitude, voire plus.
Un problème potentiel de ConGa pourrait être que la taille des tables augmente trop. Il est très facile de trouver un exemple dans lequel le premier joueur aura une table de taille exponentielle : le jeu où aucun joueur n'a de contrainte. Dans ce jeu, tous les profils de stratégies sont des PNE et sont donc enregistrés dans la table du premier joueur. Toutefois, ce comportement extrême n'a jamais été observé dans les jeux que nous avons testé en exemple. Les tables restent à des tailles raisonnables, soit parce qu'elles sont associées à des joueurs en bas de l'arbre et sont donc réinitialisées souvent, soit parce que de nombreux profils sont filtrés avant d'atteindre les joueurs des plus hauts niveaux.

Conclusion

Les Constraint Games proposent un cadre compact et naturel pour formuler des modèles en théorie des jeux. Dans cet article, nous proposons le premier solveur complet pour les constraint games basé sur un calcul de consistance par rapport aux contraintes de Nash et une détection des Never Best Responses. Nous montrons que ces techniques sont capable de surpasser le solveur Gambit considéré comme état de l'art. Mais ce travail ouvre aussi de nouvelles directions de recherche afin de trouver des techniques algorithmiques efficaces pour calculer les équilibres de Nash. Nous nous intéressons aux heuristiques de recherche, aux constraint games graphiques, et à d'autres concepts de solutions comme les équilibres de Nash Pareto-efficaces.

Remerciements. Ce travail est financé par Microsoft Research, grant MRL-2011-046.

Références

[1] Krzysztof R. Apt. Order independence and rationalizability. In Ron van der Meyden, editor, *TARK*, pages 22–38. National University of Singapore, 2005.

[2] Elise Bonzon, Marie-Christine Lagasquie-Schiex, and Jérôme Lang. Dependencies between players in boolean games. *Int. J. Approx. Reasoning*, 50(6) :899–914, 2009.

[3] Elise Bonzon, Marie-Christine Lagasquie-Schiex, Jérôme Lang, and Bruno Zanuttini. Boolean games revisited. In Gerhard Brewka, Silvia Coradeschi, Anna Perini, and Paolo Traverso, editors, *ECAI*, volume 141 of *Frontiers in Artificial Intelligence and Applications*, pages 265–269. IOS Press, 2006.

[4] Lucas Bordeaux and Brice Pajot. Computing equilibria using interval constraints. In Boi Faltings, Adrian Petcu, François Fages, and Francesca Rossi, editors, *CSCLP*, volume 3419 of *Lecture Notes in Computer Science*, pages 157–171. Springer, 2004.

[5] Mustapha Bouhtou, Guillaume Erbs, and Michel Minoux. Joint optimization of pricing and resource allocation in competitive telecommunications networks. *Networks*, 50(1) :37–49, 2007.

[6] Kenneth N. Brown, James Little, Páidí J. Creed, and Eugene C. Freuder. Adversarial constraint satisfaction by game-tree search. In Ramon López de Mántaras and Lorenza Saitta, editors, *ECAI*, pages 151–155. IOS Press, 2004.

[7] Sofie De Clercq, Kim Bauters, Steven Schockaert, Martine De Cock, and Ann Nowé. Using answer set programming for solving boolean games. In *KR*, page short paper, 2014.

[8] Marina De Vos and Dirk Vermeir. Choice logic programs and nash equilibria in strategic games. In Jörg Flum and Mario Rodríguez-Artalejo, editors, *CSL*, volume 1683 of *Lecture Notes in Computer Science*, pages 266–276. Springer, 1999.

[9] Paul E. Dunne and Wiebe van der Hoek. Representation and complexity in boolean games. In

José Júlio Alferes and João Alexandre Leite, editors, *JELIA*, volume 3229 of *Lecture Notes in Computer Science*, pages 347–359. Springer, 2004.

[10] Paul E. Dunne, Wiebe van der Hoek, Sarit Kraus, and Michael Wooldridge. Cooperative boolean games. In Lin Padgham, David C. Parkes, Jörg P. Müller, and Simon Parsons, editors, *AAMAS (2)*, pages 1015–1022. IFAAMAS, 2008.

[11] Boi Faltings. *Distributed Constraint Programming*, chapter 20, pages 699–729. Handbook of Constraint Programming. Elsevier, 2006.

[12] Norman Y. Foo, Thomas Meyer, and Gerhard Brewka. Lpod answer sets and nash equilibria. In Michael J. Maher, editor, *ASIAN*, volume 3321 of *Lecture Notes in Computer Science*, pages 343–351. Springer, 2004.

[13] Georg Gottlob, Gianluigi Greco, and Francesco Scarcello. Pure nash equilibria : Hard and easy games. *J. Artif. Intell. Res. (JAIR)*, 24 :357–406, 2005.

[14] Paul Harrenstein, Wiebe van der Hoek, John-Jules Ch. Meyer, and Cees Witteveen. Boolean Games. In Johan van Benthem, editor, *TARK*. Morgan Kaufmann, 2001.

[15] H. Hotelling. Stability in competition. *Economic Journal*, pages 41–57, 1929.

[16] Virajith Jalaparti, Giang Nguyen, Indranil Gupta, and Matthew Caesar. Cloud resource allocation games. Technical report, University of Illinois at Urbana-Champaign, 2010.

[17] Albert Xin Jiang, Kevin Leyton-Brown, and Navin A. R. Bhat. Action-graph games. *Games and Economic Behavior*, 71(1) :141–173, 2011.

[18] Michael J. Kearns, Michael L. Littman, and Satinder P. Singh. Graphical models for game theory. In Jack S. Breese and Daphne Koller, editors, *UAI*, pages 253–260. Morgan Kaufmann, 2001.

[19] Richard D. McKelvey, Andrew M. McLennan, and Theodore L. Turocy. Gambit : Software tools for game theory, 2010.

[20] J.F. Nash. Non-cooperative games. *Annals of Mathematics*, 54(2) :286–295, 1951.

[21] John Von Neumann and Oskar Morgenstern. *Theory of Games and Economic Behavior*. Princeton University Press, 1944.

[22] Thi-Van-Anh Nguyen, Arnaud Lallouet, and Lucas Bordeaux. Constraint games : Framework and local search solver. In Éric Grégoire and Bertrand Mazure, editors, *ICTAI*, Special Track on SAT and CSP Technologies, pages 963–970. Springer, 2013.

[23] Eugene Nudelman, Jennifer Wortman, Yoav Shoham, and Kevin Leyton-Brown. Run the gamut : A comprehensive approach to evaluating game-theoretic algorithms. In *AAMAS*, pages 880–887. IEEE Computer Society, 2004. `http://gamut.stanford.edu/`.

[24] M.J. Osborne and A. Rubinstein. *A Course in Game Theory*. The MIT Press, 1994.

[25] J. B. Rosen. Existence and uniqueness of equilibrium points for concave n-person games. *Econometrica*, 33(3) :520–534, July 1965.

[26] Robert W. Rosenthal. A class of games possessing pure-strategy nash equilibria. *International Journal of Game Theory*, 2(1) :65–67, 1973.

[27] Theodore L. Turocy. Personal communication, 2013.

[28] Berthold Vöcking. *Selfish load balancing*, chapter 20, pages 517–542. Algorithmic game theory. Cambridge University Press, 2007.

[29] Ling Zhao and Martin Müller. Game-SAT : A preliminary report. In Holger Hoos and David Mitchell, editors, *SAT*, pages 357–362, 2004.

Une approche CSP pour l'aide à la localisation d'erreurs*

Mohammed Bekkouche Hélène Collavizza Michel Rueher

Univ. Nice Sophia Antipolis, CNRS, I3S, UMR 7271, 06900 Sophia Antipolis, France
{helen,bekkouche,rueher}@unice.fr

Résumé

Nous proposons dans cet article une nouvelle approche basée sur la CP pour l'aide à la localisation des erreurs dans un programme pour lequel un contre-exemple est disponible, c'est à dire que l'on dispose d'une instanciation des variables d'entrée qui viole la post-condition. Pour aider à localiser les erreurs, nous générons un système de contraintes pour les chemins du CFG (Graphe de Flot de Contrôle) où au plus k instructions conditionnelles sont susceptibles de contenir des erreurs. Puis, nous calculons pour chacun de ces chemins des ensembles minima de correction (ou MCS - Minimal Correction Set) de taille bornée. Le retrait d'un de ces ensembles de contraintes produit un MSS (Maximal Satisfiable Subset) qui ne viole plus la post condition. Nous adaptons pour cela un algorithme proposé par Liffiton et Sakallah [22] afin de pouvoir traiter plus efficacement des programmes avec des calculs numériques. Nous présentons les résultats des premières expérimentations qui sont encourageants.

Abstract

We introduce in this paper a new CP-based approach to support errors location in a program for which a counter-example is available, i.e. an instantiation of the input variables that violates the post-condition. To provide helpful information for error location, we generate a constraint system for the paths of the CFG (Control Flow Graph) for which at most k conditional statements may be erroneous. Then, we calculate Minimal Correction Sets (MCS) of bounded size for each of these paths. The removal of one of these sets of constraints yields a maximal satisfiable subset, in other words, a maximal subset of constraints satisfying the post condition. We extend the algorithm proposed by Liffiton and Sakallah [22] to handle programs with numerical statements more efficiently. We present preliminary experimental results that are quite encouraging.

1 Introduction

L'aide à la localisation d'erreur à partir de contre-exemples ou de traces d'exécution est une question cruciale lors de la mise au point de logiciels critiques. En effet, quand un programme P contient des erreurs, un model-checker fournit un contre-exemple ou une trace d'exécution qui est souvent longue et difficile à comprendre, et de ce fait d'un intérêt très limité pour le programmeur qui doit débugger son programme. La localisation des portions de code qui contiennent des erreurs est donc souvent un processus difficile et coûteux, même pour des programmeurs expérimentés. C'est pourquoi nous proposons dans cet article une nouvelle approche basée sur la CP pour l'aide à la localisation des erreurs dans un programme pour lequel un contre-exemple a été trouvé ; c'est à dire pour lequel on dispose d'une instanciation des variables d'entrée qui viole la post-condition. Pour aider à localiser les erreurs, nous générons un système de contraintes pour les chemins du CFG (Graphe de Flot de Contrôle) où au plus k instructions conditionnelles sont susceptibles de contenir des erreurs. Puis, nous calculons pour chacun de ces chemins des ensembles minima de correction (ou MCS - Minimal Correction Set) de taille bornée. Le retrait d'un de ces ensembles des contraintes initiales produit un MSS (Maximal Satisfiable Subset) qui ne viole plus la post condition. Nous adaptons pour cela un algorithme proposé par Liffiton et Sakallah afin de pouvoir traiter plus efficacement des programmes avec des calculs numériques. Nous présentons les résultats des premières expérimentations qui sont encourageants.

La prochaine section est consacrée à un positionnement de notre approche par rapport aux principales méthodes qui ont été proposées pour ce résoudre ce problème. La section suivante est dédiée à la description de notre approche et des algorithmes utilisés. Puis, nous présentons les résultats des premières expérimentations avant de conclure.

2 État de l'art

Dans cette section, nous positionnons notre approche par rapport aux principales méthodes existantes. Nous parlerons d'abord des méthodes utilisées pour l'aide à la localisation des erreurs dans le cadre du test et de la vérification de programmes. Comme

*Ce travail a débuté au NII (National Institute of Informatics, 2-1-2 Hitotsubashi, Chiyoda-ku, Tokyo 101-8430) où Michel Rueher a été invité plusieurs fois en tant que Professeur associé depuis 2011. Les idées générales et la démarche ont été définies lors des réunions de travail avec les professeurs Hiroshi HOSOBE et Shin NAKAJIMA.

l'approche que nous proposons consiste essentiellement à rechercher des sous-ensembles de contraintes spécifiques dans un système de contraintes inconsistant, nous parlerons aussi dans un second temps des algorithmes qui ont été utilisés en recherche opérationnelle et en programmation par contraintes pour aider l'utilisateur à debugger un système de contraintes inconsistant.

2.1 Méthodes d'aide à la localisation des erreurs utilisées en test et en vérification de programmes

Différentes approches ont été proposées pour aider le programmeur dans l'aide à la localisation d'erreurs dans la communauté test et vérification.

Le problème de la localisation des erreurs a d'abord été abordé dans le cadre du test où de nombreux systèmes ont été développés. Le plus célèbre est Tarantula [16, 15] qui utilise différentes métriques pour établir un classement des instructions suspectes détectées lors de l'exécution d'une batterie de tests. Le point critique de cette approche réside dans le fait qu'elle requiert un oracle qui permet de décider si le résultat du test est juste ou non. Nous nous plaçons ici dans un cadre moins exigeant (et plus réaliste) qui est celui du Bounded-Model Checking (BMC), c'est à dire un cadre où les seuls prérequis sont un programme, une post-condition ou une assertion qui doit être vérifiée, et éventuellement une pré-condition.

C'est aussi dans ce cadre que se placent Bal et al[1] qui utilisent plusieurs appels à un Model Checker et comparent les contre-exemples obtenus avec une trace d'exécution correcte. Les transitions qui ne figurent pas dans la trace correcte sont signalées comme une possible cause de l'erreur. Ils ont implanté leur algorithme dans le contexte de SLAM, un model checker qui vérifie les propriétés de sécurité temporelles de programmes C.

Plus récemment, des approches basées sur la dérivation de traces correctes ont été introduites dans un système nommé `Explain`[14, 13] et qui fonctionne en trois étapes :

1. Appel de `CBMC` [1] pour trouver une exécution qui viole la post-condition ;
2. Utilisation d'un solveur pseudo-booléen pour rechercher l'exécution correcte la plus proche ;
3. Calcul de la différence entre les traces.

`Explain` produit ensuite une formule propositionnelle S associée à `P` mais dont les affectations ne violent pas la spécification. Enfin `Explain` étend S avec des contraintes représentant un problème d'optimisation : trouver une affectation satisfaisante qui soit aussi proche que possible du contre-exemple ; la proximité étant mesurée par une distance sur les exécutions de `P`.

Une approche similaire à `Explain` a été introduite dans [25] mais elle est basée sur le test plutôt que sur la vérification de modèles : les auteurs utilisent des séries de tests correctes et des séries erronées. Ils utilisent aussi des métriques de distance pour sélectionner un test correct à partir d'un ensemble donné de tests. Cette approche suppose qu'un oracle soit disponible.

Dans [11, 12], les auteurs partent aussi de la trace d'un contre-exemple, mais ils utilisent la spécification pour dériver un programme correct pour les mêmes données d'entrée. Chaque instruction identifiée est un candidat potentiel de faute et elle peut être utilisée pour corriger les erreurs. Cette approche garantit que les erreurs sont effectivement parmi les instructions identifiées (en supposant que l'erreur est dans le modèle erroné considéré). En d'autres termes, leur approche identifie un sur-ensemble des instructions erronées. Pour réduire le nombre d'erreurs potentielles, le processus est redémarré pour différents contre-exemples et les auteurs calculent l'intersection des ensembles d'instructions suspectes. Cependant, cette approche souffre de deux problèmes majeurs :

- Elle permet de modifier n'importe quelle expression, l'espace de recherche peut ainsi être très grand ;
- Elle peut renvoyer beaucoup de faux diagnostics totalement absurdes car toute modification d'expression est possible (par exemple, changer la dernière affectation d'une fonction pour renvoyer le résultat attendu).

Pour remédier à ces inconvénients, Zhang et al [28] proposent de modifier uniquement les prédicats de flux de contrôle. L'intuition de cette approche est qu'à travers un switch des résultats d'un prédicat et la modification du flot de contrôle, l'état de programme peut non seulement être modifié à peu de frais, mais qu'en plus, il est souvent possible d'arriver à un état succès. Liu et al [23] généralisent cette approche en permettant la modification de plusieurs prédicats. Ils proposent également une étude théorique d'un algorithme de débogage pour les erreurs *RHS*, c'est à dire les erreurs dans les prédicats de contrôle et la partie droite des affectations.

Dans [3], les auteurs abordent le problème de l'analyse de la trace d'un contre-exemple et de l'identification de l'erreur dans le cadre des systèmes de vérification formelle du hardware. Ils utilisent pour cela la notion de causalité introduite par Halpern et Pearl pour définir formellement une série de causes de la violation de la spécification par un contre-exemple.

Récemment, Manu Jose et Rupak Majumdar [17, 18] ont abordé ce problème différemment : ils ont introduit un nouvel algorithme qui utilise un solveur MAX-SAT pour calculer le nombre maximum des clauses d'une formule booléenne qui peut être satisfaite par une affectation. Leur algorithme fonctionne en trois étapes :

1. ils encodent une trace d'un programme par une formule booléenne F qui est satisfiable si et seulement si la trace est satisfiable ;
2. ils construisent une formule fausse F' en imposant que la post-condition soit vraie (la formule F' est insatisfiable car la trace correspond à un contre-exemple qui viole la post-condition) ;
3. Ils utilisent `MAXSAT` pour calculer le nombre

1. `http://www.cprover.org/cbmc/`

maximum de clauses pouvant être satisfaites dans F' et affichent le complément de cet ensemble comme une cause potentielle des erreurs. En d'autres termes, ils calculent le complément d'un MSS (Maximal Satisfiable Subset).

Manu Jose et Rupak Majumdar [17, 18] ont implanté leur algorithme dans un outil appelé `BugAssist` qui utilise `CBMC`.

Si-Mohamed Lamraoui et Shin Nakajima [20] ont aussi développé récemment un outil nommé `SNIPER` qui calcule les MSS d'une formule $\psi = EI \wedge TF \wedge AS$ où EI encode les valeurs d'entrée erronées, TF est une formule qui représente tous les chemins du programme, et AS correspond à l'assertion qui est violée. Les MCS sont obtenus en prenant le complément des MSS calculés. L'implémentation est basée sur la représentation intermédiaire `LLVM` et le solveur SMT `Yices`. L'implémentation actuelle est toutefois beaucoup plus lente que `BugAssist`.

L'approche que nous proposons ici est inspirée des travaux de Manu Jose et Rupak Majumdar. Les principales différences sont :

1. Nous ne transformons pas tout le programme en un système de contraintes mais nous utilisons le graphe de flot de contrôle pour collecter les contraintes du chemin du contre exemple et des chemins dérivés de ce dernier en supposant qu'au plus k instructions conditionnelles sont susceptibles de contenir des erreurs.
2. Nous n'utilisons pas des algorithmes basés sur `MAXSAT` mais des algorithmes plus généraux qui permettent plus facilement de traiter des contraintes numériques.

2.2 Méthodes pour débugger un système de contraintes inconsistant

En recherche opérationnelle et en programmation par contraintes, différents algorithmes ont été proposés pour aider l'utilisateur à debugger un système de contraintes inconsistant. Lorsqu'on recherche des informations utiles pour la localisation des erreurs sur les systèmes de contraintes numériques, on peut s'intéresser à deux types d'informations :

1. Combien de contraintes dans un ensemble de contraintes insatisfiables peuvent être satisfaites ?
2. Où se situe le problème dans le système de contraintes ?

Avant de présenter rapidement les algorithmes [2] qui cherchent à répondre à ces questions, nous allons définir plus formellement les notion de MUS, MSS et MCS à l'aide des définitions introduites dans [22].
Un MUS (Minimal Unsatisfiable Subsets) est un ensemble de contraintes qui est inconsistant mais qui devient consistant si une contrainte quelconque est retirée de cet ensemble. Plus formellement, soit C un ensemble de contraintes :

$$M \subseteq C \text{ est un MUS} \Leftrightarrow M \text{ est UNSAT}$$
$$\text{et } \forall c \in M : M \setminus \{c\} \text{ est SAT.}$$

La notion de MSS (Maximal Satisfiable Subset) est une généralisation de MaxSAT / MaxCSP où l'on considère la maximalité au lieu de la cardinalité maximale :

$$M \subseteq C \text{ est un MSS} \Leftrightarrow M \text{ est SAT}$$
$$\text{et } \forall c \in C \setminus M : M \cup \{c\} \text{ est UNSAT.}$$

Cette définition est très proche de celle des IIS (Irreducible Inconsistent Subsystem) utilisés en recherche opérationnelle [5, 6, 7].
Les MCS (Minimal Correction Set) sont des compléments des MSS (le retrait d'un MCS à C produit un MSS car on "corrige" l'infaisabilité) :

$$M \subseteq C \text{ est un MCS} \Leftrightarrow C \setminus M \text{ est SAT}$$
$$\text{et } \forall c \in M : (C \setminus M) \cup \{c\} \text{ est UNSAT.}$$

Il existe donc une dualité entre l'ensemble des MUS et des MCS [4, 22] : informellement, l'ensemble des MCS est équivalent aux ensembles couvrants irréductibles [3] des MUS ; et l'ensemble des MUS est équivalent aux ensembles couvrants irréductibles des MCS. Soit un ensemble de contraintes C :

1. Un sous-ensemble M de C est un MCS ssi M est un ensemble couvrant minimal des MUS de C ;
2. Un sous-ensemble M de C est un MUS ssi M est un ensemble couvrant minimal des MCS de C ;

Au niveau intuitif, il est aisé de comprendre qu'un MCS doit au moins retirer une contrainte de chaque MUS. Et comme un MUS peut être rendu satisfiable en retirant n'importe laquelle de ses contraintes, chaque MCS doit au moins contenir une contrainte de chaque MUS. Cette dualité est aussi intéressante pour notre problématique car elle montre que les réponses aux deux questions posées ci-dessus sont étroitement liées.

Différents algorithmes ont été proposés pour le calcul des IIS/MUS et MCS. Parmi les premiers travaux, ont peut mentionner les algorithmes `Deletion Filter`, `Additive Method`, `Additive Deletion Method`, `Elastic Filter` qui ont été développés dans la communauté de recherche opérationnelle [5, 27, 6, 7]. Les trois premiers algorithmes sont des algorithmes itératifs alors que le quatrième utilise des variables d'écart pour identifier dans la première phase du Simplexe les contraintes susceptibles de figurer dans un IIS.
Junker [19] a proposé un algorithme générique basé sur une stratégie "Divide-and-Conquer" pour calculer efficacement les IIS/MUS lorsque la taille des sous-ensembles conflictuels est beaucoup plus petite que celle de l'ensemble total des contraintes.
L'algorithme de Liffiton et Sakallah [22] qui calcule d'abord l'ensemble des MCS par ordre de taille croissante, puis l'ensemble des MUS est basé sur la propriété mentionnée ci-dessus. Cet algorithme, que nous avons utilisé dans notre implémentation est décrit dans la section suivante.

Différentes améliorations [10, 21, 24] de ces algorithmes ont été proposées ces dernières années mais elles sont assez étroitement liées à SAT et a priori assez

2. Pour une présentation plus détaillée voir `http://users.polytech.unice.fr/~rueher/Publis/Talk_NII_2013-11-06.pdf`

3. Soit Σ un ensemble d'ensemble et D l'union des élements de Σ. On rappelle que H est un ensemble couvrant de Σ si $H \subseteq D$ et $\forall S \in \Sigma : H \cup S \neq \emptyset$. H est irréductible (ou minimal) si aucun élément ne peut être retiré de H sans que celui-ci ne perde sa propriété d'ensemble couvrant.

difficilement transposables dans un contexte où nous avons de nombreuses contraintes numériques.

3 Notre approche

Dans cette section nous allons d'abord présenter le cadre général de notre approche, à savoir celui du "Bounded Model Checking" (BMC) basé sur la programmation par contraintes, puis nous allons décrire la méthode proposée et les algorithmes utilisés pour calculer des MCS de cardinalité bornée.

3.1 Les principes : BMC et MCS

Notre approche se place dans le cadre du "Bounded model Checking" (BMC) par programmation par contraintes [8, 9]. En BMC, les programmes sont dépliés en utilisant une borne b, c'est à dire que les boucles sont remplacées par des imbrications de conditionnelles de profondeur au plus b. Il s'agit ensuite de détecter des non-conformités par rapport à une spécification. Étant donné un triplet de Hoare $\{PRE,PROG_b,POST\}$, où PRE est la pré-condition, $PROG_b$ est le programme déplié b fois et $POST$ est la post-condition, le programme est *non conforme* si la formule $\Phi = PRE \wedge PROG_b \wedge \neg POST$ est satisfiable. Dans ce cas, une instanciation des variables de Φ est un *contre-exemple*, et un cas de non conformité, puisqu'il satisfait à la fois la pré-condition et le programme, mais ne satisfait pas la post-condition.

`CPBPV` [8] est un outil de BMC basé sur la programmation par contraintes. $CPBPV$ transforme PRE et $POST$ en contraintes, et transforme $PROG_b$ en un CFG dans lequel les conditions et les affectations sont traduites en contraintes [4]. `CPBPV` construit le `CSP` de la formule Φ *à la volée*, par un parcours en profondeur du graphe. À l'état initial, le `CSP` contient les contraintes de PRE et $\neg POST$, puis les contraintes d'un chemin sont ajoutées au fur et à mesure de l'exploration du graphe. Quand le dernier noeud d'un chemin est atteint, la faisabilité du `CSP` est testée. S'il est consistant, alors on a trouvé un contre-exemple, sinon, un retour arrière est effectué pour explorer une autre branche du CFG. Si tous les chemins ont été explorés sans trouver de contre-exemple, alors le programme est conforme à sa spécification (sous réserve de l'hypothèse de dépliage des boucles).

Les travaux présentés dans cet article cherchent à *localiser* l'erreur détectée par la phase de BMC. Plus précisemment, soit CE une instanciation des variables qui satisfait le `CSP` contenant les contraintes de PRE et $\neg POST$, et les contraintes d'un chemin incorrect de $PROG_b$ noté $PATH$. Alors le `CSP` $C = CE \cup PRE \cup PATH \cup POST$ est *inconsistant*, puisque CE est un contre-exemple et ne satisfait donc pas la post-condition. Un *ensemble minima de correction* (ou MCS - Minimal Correction Set) de C est un ensemble de contraintes qu'il faut nécessairement enlever pour que C devienne consistant. Un tel ensemble fournit donc une *localisation de l'erreur* sur le chemin du contre-exemple. Comme l'erreur peut se trouver dans une affectation sur le chemin du contre-exemple, mais peut aussi provenir d'un mauvais branchement, notre approche (nommée `LocFaults`) s'intéresse également aux MCS des systèmes de contraintes obtenus en déviant des branchements par rapport au comportement induit par le contre-exemple. Plus précisément, l'algorithme $LocFaults$ effectue un parcours en profondeur d'abord du CFG de $PROG_b$, en propageant le contre-exemple et en déviant au plus k_{max} conditions. Trois cas peuvent être distingués :

- Aucune condition n'a été déviée : `LocFaults` a parcouru le chemin du contre-exemple en collectant les contraintes de ce chemin et il va calculer les MCS sur cet ensemble de contraintes ;
- k_{max} conditions ont été déviées sans qu'on arrive à trouver un chemin qui satisfasse la post-condition : on abandonne l'exploration de ce chemin ;
- d conditions on déja été déviées et on peut encore dévier au moins une condition, c'est à dire $k_{max} > 1$. Alors la condition courante c est déviée. Si le chemin résultant ne viole plus la post-condition, l'erreur sur le chemin initial peut avoir deux causes différentes :
  (i) les conditions déviées elles-mêmes sont cause de l'erreur,
  (ii) une erreur dans une affectation a provoqué une mauvaise évaluation de c, faisant prendre le mauvais branchement.
  Dans le cas (ii), le `CSP` $CE \cup PRE \cup PATH_c \cup \{c\}$, où $PATH_c$ est l'ensemble des contraintes du chemin courant, c'est à dire le chemin du contre-exemple dans lequel d déviations ont déjà été prises, est satisfiable. Par conséquent, le `CSP` $CE \cup PRE \cup PATH_c \cup \{\neg c\}$ est *insatisfiable*. `LocFaults` calcule donc également les MCS de ce `CSP`, afin de détecter les instructions suspectées d'avoir induit le mauvais branchement pour c.
  Bien entendu, nous ne calculons pas les MCS des chemins ayant le même préfixe : si la déviation d'une condition permet de satisfaire la post-condition, il est inutile de chercher à modifier des conditions supplémentaires dans la suite du chemin

3.2 Description de l'algorithme

L'algorithme `LocFaults` (cf. Algorithm 1) prend en entrée un programme déplié non conforme vis-à-vis de sa spécification, un contre-exemple, et une borne maximum de la taille des ensembles de correction. Il dévie au plus k_{max} conditions par rapport au contre-exemple fourni, et renvoie une liste de corrections possibles.

`LocFaults` commence par construire le CFG du programme puis appelle la fonction `DFS` sur le chemin du contre-exemple (i.e. en déviant 0 condition) puis en acceptant au plus k_{max} déviations. La fonction `DFS` gère trois ensembles de contraintes :

- CSP_d : l'ensemble des contraintes des conditions

4. Pour éviter les problèmes de re-définitions multiples des variables, la forme DSA (Dynamic Single Assignment [2]) est utilisée

Algorithm 1: LocFaults

```
1  Fonction LocFaults(PROG_b,CE,k_max,MCS_b)
   Entrées:
   – PROG_b : un programme déplié b fois non conforme vis-à-vis de sa spécification,
   – CE : un contre-exemple de PROG_b,
   – k_max : le nombre maximum de conditions à dévier,
   – MCS_b : la borne du cardinal des MCS
   Sorties: une liste de corrections possibles
2  début
3      CFG ← CFG_build(PROG_b) % construction du CFG
4      MCS = []
5      DFS_devie(CFG.root,CE,∅,∅,0,MCS,MCS_b) % calcul des MCS sur le chemin du contre-exemple
6      DFS_devie(CFG.root,CE,∅,∅,k_max,MCS,MCS_b) % calcul des MCS en prenant au plus k_max déviations
7      retourner MCS
8  fin
```

Algorithm 2: DFS_{devie}

```
1  Fonction DFS_devie(n,P,CSP_d,CSP_a,k,MCS,MCS_b)
   Entrées:
   – n : noeud du CFG,
   – P : contraintes de propagation (issues du contre-exemple et du chemin),
   – CSP_d : contraintes des conditions déviées,
   – CSP_a : contraintes des affectations,
   – k : nombre de conditions à dévier,
   – MCS : ensemble des MCS calculés,
   – MCS_b : la borne du cardinal des MCS
2  début
3      si n est la postcondition alors
4          % on est sur le chemin du CE, calcul des MCS
5          CSP_a ← CSP_a ∪ {cstr(POST)}
6          MCS.add(MCS(CSP_a, MCS_b))
7      fin
8      sinon si n est un noeud conditionnel alors
9          si P ∪ {cstr(n.cond)} est faisable alors
10             % next est le noeud où l'on doit aller, devie est la branche opposée
11             next = n.gauche
12             devie = n.droite
13         fin
14         sinon
15             next = n.droite
16             devie = n.gauche
17         fin
18         si k > 0 alors
19             % on essaie de dévier la condition courante
20             corrige = correct(devie,P)
21             si corrige alors
22                 % le chemin est corrigé, on met à jour les MCS
23                 CSP_d ← CSP_d ∪ {cstr(n.cond)}
24                 MCS.addAll(CSP_d) % ajout des conditions déviées
25                 % calcul des MCS sur le chemin qui mène à la dernière condition déviée
26                 pour chaque c dans CSP_d faire
27                     CSP_a ← CSP_a ∪ {¬c}
28                 fin
29                 MCS.add(MCS(CSP_a, MCS_b))
30             fin
31             sinon si k>1 alors
32                 % on essaie de dévier la condition courante et des conditions en dessous
33                 DFS_devie(devie,P,CSP_d ∪ {cstr(n.cond)},CSP_a,k − 1,MCS,MCS_b)
34             fin
35             % dans tous les cas, on essaie de dévier les conditions en dessous du noeud courant
36             DFS_devie(next,P,CSP_d,CSP_a,k,MCS,MCS_b)
37         fin
38         sinon
39             % k=0, on est sur le chemin du contre-exemple, on suit le chemin
40             DFS_devie(next,P,CSP_d,CSP_a,k,MCS,MCS_b)
41         fin
42     fin
43     sinon si (n est un bloc d'affectations) alors
44         pour chaque affectation ass ∈ n.assigns faire
45             P.add(propagate(ass,P))
46             CSP_a ← CSP_a ∪ {cstr(ass)}
47         fin
48         % On continue l'exploration sur le noeud suivant
49         DFS_devie(n.next,CE,P,CSP_d,CSP_a,k,MCS,MCS_b)
50     fin
51 fin
```

qui ont été déviées à partir du chemin du contre-exemple,

- CSP_a : l'ensemble des contraintes d'affectations du chemin,
- P : l'ensemble des contraintes dites de propagation, c'est à dire les contraintes de la forme *variable* = *constante* qui sont obtenues en propageant le contre exemple sur les affectations du

Algorithm 3: correct

```
1  Fonction correct(n,P)
   Entrées:
   – n : noeud du CFG,
   – P : contraintes de propagation (issues du contre-exemple et
     du chemin),
   Sorties: true si le programme est correct sur le chemin
            induit par P
2  début
3      si n est la postcondition alors
4          si P ∪ {cstr(POST)} est faisable alors
5              retourner true
6          fin
7          sinon
8              retourner false
9          fin
10     fin
11     sinon si n est un noeud conditionnel alors
12         si P ∪ {cstr(n.cond)} est faisable alors
13             % exploration de la branche If
14             retourner correct(n.left,P)
15         fin
16         sinon
17             retourner correct(n.right,P)
18         fin
19     fin
20     sinon si (n est un bloc d'affectations) alors
21         % on propage les affectations
22         pour chaque affectation ass ∈ n.assigns faire
23             P.add(propagate(ass,P))
24         fin
25         % On continue l'exploration sur le noeud suivant
26         retourner
           correct(n.next,P,CSP_d,CSP_a,MCS,MCS_b)
27     fin
28 fin
```

Algorithm 4: MCS

```
1  Fonction MCS(C,MCS_b)
   Entrées: C : Ensemble de contraintes infaisable,
            MCS_b : Entier
   Sorties: MCS : Liste de MCS de C de
            cardinalité inférieure à MCS_b
2  début
3      C' ← AddYVars(C)
4      MCS ← ∅
5      k ← 1
6      tant que SAT(C') ∧ k<=MCS_b faire
7          C'_k ← C' ∧
           AtMost({¬y_1, ¬y_2, ..., ¬y_n},k)
8          tant que SAT(C'_k) faire
9              MCS.add(newMCS).
10             C'_k ← C'_k ∧
               BlockingClause(newMCS)
11             C' ← C' ∧
               BlockingClause(newMCS).
12         fin
13         k ← k + 1.
14     fin
15     retourner MCS
16 fin
```

chemin.

L'ensemble P est utilisé pour propager les informations et vérifier si une condition est satisfaite, les ensembles CSP_d et CSP_a sont utilisés pour calculer les MCS. Ces trois ensembles sont collectés à la volée lors du parcours en profondeur. Les paramètres de la fonction `DFS` sont les ensembles CSP_d, CSP_a et P décrits ci-dessus, n le noeud courant du CFG, MCS la liste des corrections en cours de construction, k le nombre de déviations autorisées et MCS_b la borne de la taille des MCS. Nous notons $n.left$ (resp. $n.right$) la branche *if* (resp. *else*) d'un noeud conditionnel, et $n.next$ le noeud qui suit un bloc d'affectation ; *cstr* est la fonction qui traduit une condition ou affectation en contraintes.

Le parcours commence avec CSP_d et CSP_a vides et P contenant les contraintes du contre-exemple. Il part de la racine de l'arbre ($CFG.root$) qui contient la pré-condition, et se déroule comme suit :

- Quand le dernier noeud est atteint (i.e. noeud de la post-condition), on est sur le chemin du contre-exemple. La post-condition est ajoutée à CSP_a et on cherche les MCS,
- Quand le noeud est un noeud conditionnel, alors on utilise P pour savoir si la condition est satisfaite. Si on peut encore prendre une déviation (i.e. $k > 0$), on essaie de dévier la condition courante c et on vérifie si cette déviation corrige le programme en appelant la fonction *correct*. Cette fonction propage tout simplement le contre-exemple sur le graphe à partir du noeud courant et renvoie vrai si le programme satisfait la post-condition pour ce chemin,
  - Si dévier c a corrigé le programme, alors les conditions qui ont été déviées (i.e. $CSP_d \cup c$) sont des corrections. De plus, on calcule aussi les corrections dans le chemin menant à c,
  - Si dévier c n'a pas corrigé le programme, si on peut encore dévier des conditions (i.e. $k \geq 1$) alors on dévie c et on essaie de dévier $k-1$ conditions en dessous de c.

  Dans les deux cas (dévier c a corrigé ou non le programme), on essaie aussi de dévier des conditions en dessous de c, sans dévier c.
- Quand le noeud est un bloc d'affectations, on propage le contre-exemple sur ces affectations et on ajoute les contraintes correspondantes dans P et dans CSP_a.

L'algorithme `LocFaults` appelle l'algorithme `MCS` (cf. Algorithm 4) qui est une transcription directe de l'algorithme proposé par Liffiton et Sakallah [22]. Cet algorithme associe à chaque contrainte un sélecteur de variable y_i qui peut prendre la valeur 0 ou 1 ; la contrainte `AtMost` permet donc de retenir au plus k contraintes du système de contraintes initial dans le MCS. La procédure `BlockingClause(newMCS)` appelée à la ligne 10 (resp. ligne 11) permet d'exclure les sur-ensembles de taille k (resp. de taille supérieure à k).

Lors de l'implémentation de cet algorithme nous avons utilisé `IBM ILOG CPLEX`[5] qui permet à la fois une implémentation aisée de la fonction `AtMost` et la résolution de systèmes de contraintes numériques. Il faut toutefois noter que cette résolution n'est correcte que sur les entiers et que la prise en compte des nombres flottants nécessite l'utilisation d'un solveur spécialisé pour le traitement des flottants.

5. `http://www-01.ibm.com/software/commerce/optimization/cplex-optimizer/`

Programme	Contre-exemple	Erreurs	LocFaults				BugAssist
			$= 0$	≤ 1	≤ 2	≤ 3	
AbsMinusKO	$\{i = 0, j = 1\}$	17	{17}	{17}	{17}	{17}	{17}
AbsMinusKO2	$\{i = 0, j = 1\}$	11	{11},{17}	{11},{17}	{11},{17}	{11},{17}	{17, 20, 16}
AbsMinusKO3	$\{i = 0, j = 1\}$	14	{20}	{16},{14},{12}, {20}	{16},{14},{12}, {20}	{16},{14},{12}, {20}	{16, 20}
MinmaxKO	$\{in_1 = 2, in_2 = 1, in_3 = 3\}$	19	{10},{19}	{18},{10} {10},{19}	{18},{10} {10},{19}	{18},{10} {10},{19}	{18, 19, 22}
MidKO	$\{a = 2, b = 1, c = 3\}$	19	{19}	{19}	{19}	{14, 23, 26} {19}	{14, 19, 30}
Maxmin6varKO	$\{a = 1, b = -4, c = -3, d = -1, e = 0, f = -4\}$	27	{28}	{15} {27} {28}	{15} {27} {28}	{15} {27} {28}	{15, 12, 27, 31, 166}
Maxmin6varKO2	$\{a = 1, b = -3, c = 0, d = -2, e = -1, f = -2\}$	12	{65}	{12} {65}	{12} {65}	{12} {65}	{12, 64, 166}
Maxmin6varKO3	$\{a = 1, b = -3, c = 0, d = -2, e = -1, f = -2\}$	12,15	{65}	{65}	{12, 15} {65}	{12, 15} {65}	{12, 15, 64, 166}
Maxmin6varKO4	$\{a = 1, b = -3, c = -4, d = -2, e = -1, f = -2\}$	12,15, 19	{116}	{116}	{116}	{12, 15, 19} {116}	{12, 166}
TritypeKO	$\{i = 2, j = 3, k = 2\}$	54	{54}	{26} {48} {30},{25} {54}	{26} {29, 32} {48},{30},{25} {53, 57},{25},{30} {54}	{26} {29, 32} {29, 35, 57},{25} {32, 44, 57},{33}, {25},{30} {48},{30},{25} {53, 57},{25},{30} {54}	{26, 27, 32, 33, 36, 48, 57, 68}
TritypeKO2	$\{i = 2, j = 2, k = 4\}$	53	{54}	{21} {26} {35},{27},{25} {53},{25},{27} {54}	{21} {26} {29, 57},{30},{27}, {25} {32, 44},{33},{25}, {27} {35},{27},{25} {53},{25},{27} {54}	{21} {26} {29, 57},{30},{27}, {25} {32, 44},{33},{25}, {27} {35},{27},{25} {53},{25},{27} {54}	{21, 26, 27, 29, 30, 32, 33, 35, 36, 53, 68}
TritypeKO2V2	$\{i = 1, j = 2, k = 1\}$	31	{50}	{21} {26} {29} {36},{31},{25} {49},{31},{25} {50}	{21} {26} {29} {33, 45},{34},{31}, {25} {36},{31},{25} {49},{31},{25} {50}	{21} {26} {29} {33, 45},{34},{31}, {25} {36},{31},{25} {49},{31},{25} {50}	{21, 26, 27, 29, 31, 33, 34, 36, 37, 49, 68}
TritypeKO3	$\{i = 1, j = 2, k = 1\}$	53	{54}	{21} {29} {35},{30},{25} {53},{30},{25} {54}	{21} {26, 57},{30},{25}, {27} {29} {32, 44},{33},{30}, {25} {35},{30},{25} {53},{30},{25} {54}	{21} {26, 57},{30},{25}, {27} {29} {32, 44},{33},{30}, {25} {35},{30},{25} {53},{30},{25} {54}	{21, 26, 27, 29, 30, 32, 33, 35, 36, 48, 53, 68}
TritypeKO4	$\{i = 2, j = 3, k = 3\}$	45	{46}	{45},{33},{25} {46}	{26, 32} {29, 32} {45},{33},{25} {46}	{26, 32} {29, 32} {32, 35, 49},{25} {32, 35, 53},{25} {32, 35, 57},{25} {45},{33},{25} {46}	{26, 27, 29, 30, 32, 33, 35, 45, 49, 68}
TritypeKO5	$\{i = 2, j = 3, k = 3\}$	32,45	{40}	{26} {29} {40}	{26} {29} {32, 45},{33},{25} {35, 49},{25} {35, 53},{25} {35, 57},{25} {40}	{26} {29} {32, 45},{33},{25} {35, 49},{25} {35, 53},{25} {35, 57},{25} {40}	{26, 27, 29, 30, 32, 33, 35, 49, 68}
TritypeKO6	$\{i = 2, j = 3, k = 3\}$	32,33	{40}	{26} {29} {40}	{26} {29} {35, 49},{25} {35, 53},{25} {35, 57},{25} {40}	{26} {29} {32, 45, 49},{33},{25} {32, 45, 53},{33},{25} {32, 45, 57},{33},{25} {35, 49},{25} {35, 53},{25} {35, 57},{25} {40}	{26, 27, 29, 30, 32, 33 35, 49, 68}
TriPerimetreKO	$\{i = 2, j = 1, k = 2\}$	58	{58}	{31} {37},{32},{27} {58}	{31} {37},{32},{27} {58}	{31} {37},{32},{27} {58}	{28, 29, 31, 32, 35, 37, 65, 72}
TriPerimetreKOV2	$\{i = 2, j = 3, k = 2\}$	34	{60},{34}	{32} {40},{33},{27} {60}, {34}	{32} {40},{33},{27} {60}, {34}	{32} {40},{33},{27} {60}, {34}	{28, 32, 33, 34, 36, 38, 40, 41, 52, 55, 56, 60, 64, 67, 74}

TABLE 1 – MCS identifiés par LocFaults pour des programmes sans boucles

4 Evaluation expérimentale

Pour évaluer la méthode que nous avons proposée, nous avons comparé les performances de `LocFaults` et de `BugAssist` [17, 18] sur un ensemble de programmes. Comme `LocFaults` est basé sur `CPBPV`[8] qui travaille sur des programmes Java et que `BugAssist` travaille sur des programmes C, nous avons construit pour chacun des programmes :

- une version en Java annotée par une spécification JML ;
- une version en ANSI-C annotée par la même spécification mais en ACSL.

Les deux versions ont les mêmes numéros de ligne et les mêmes instructions. La précondition est un contre-exemple du programme, et la postcondition correspond au résultat de la version correcte du programme pour les données du contre-exemple. Nous avons considéré qu'au plus trois conditions pouvaient être fausses sur un chemin. Par ailleurs, nous n'avons pas cherché de MCS de cardinalité supérieure à 3. Les expérimentations ont été effectuées avec un processeur Intel Core i7-3720QM 2.60 GHz avec 8 GO de RAM.

Nous avons d'abord utilisé un ensemble de programmes académiques de petite taille (entre 15 et 100 lignes). A savoir :

- **AbsMinus**. Ce programme prend en entrée deux entiers i et j et renvoie la valeur absolue de $i - j$.
- **Minmax**. Ce programme prend en entrée trois entiers : $in1$, $in2$ et $in3$, et permet d'affecter la plus petite valeur à la variable *least* et la plus grande valeur à la variable *most*.

Programme	LocFaults					BugAssist	
	P	L				P	L
		$= 0$	≤ 1	≤ 2	≤ 3		
AbsMinusKO	$0,487s$	$0,044s$	$0,073s$	$0,074s$	$0,062s$	$0,02s$	$0,03s$
AbsMinusKO2	$0,484s$	$0,085s$	$0,065s$	$0,085s$	$0,078s$	$0,01s$	$0,06s$
AbsMinusKO3	$0,479s$	$0,076s$	$0,113s$	$0,357s$	$0,336s$	$0,02s$	$0,04s$
MinmaxKO	$0,528s$	$0,243s$	$0,318s$	$0,965s$	$1,016s$	$0,01s$	$0,09s$
MidKO	$0,524s$	$0,065s$	$0,078s$	$0,052s$	$0,329s$	$0,02s$	$0,08s$
Maxmin6varKO	$0,528s$	$0,082s$	$0,132s$	$0,16s$	$0,149s$	$0,06s$	$1,07s$
Maxmin6varKO2	$0,536s$	$0,064s$	$0,072s$	$0,097s$	$0,126s$	$0,06s$	$0,66s$
Maxmin6varKO3	$0,545s$	$0,066s$	$0,061s$	$0,29s$	$0,307s$	$0,04s$	$1,19s$
Maxmin6varKO4	$0,538s$	$0,06s$	$0,07s$	$0,075s$	$0,56s$	$0,04s$	$0,78s$
TritypeKO	$0,493s$	$0,022s$	$0,097s$	$0,276s$	$2,139s$	$0,03s$	$0,35s$
TritypeKO2	$0,51s$	$0,023s$	$0,25s$	$2,083$	$3,864s$	$0,02s$	$0,69s$
TritypeKO2V2	$0,514s$	$0,034s$	$0,28s$	$1,178s$	$1,31s$	$0,02s$	$0,77s$
TritypeKO3	$0,493s$	$0,022s$	$0,26s$	$1,928s$	$4,535s$	$0,02$	$0,48s$
TritypeKO4	$0,497s$	$0,023s$	$0,095s$	$0,295$	$5,127s$	$0,02s$	$0,21s$
TritypeKO5	$0,492s$	$0,021s$	$0,099s$	$0,787s$	$0,8s$	$0,01s$	$0,25s$
TritypeKO6	$0,492s$	$0,025s$	$0,078s$	$0,283s$	$1,841s$	$0,03s$	$0,24s$
TriPerimetreKO	$0,518s$	$0,047s$	$0,126s$	$1,096s$	$2,389s$	$0,03s$	$0,64s$
TriPerimetreKOV2	$0,503s$	$0,043s$	$0,271s$	$0,639s$	$1,958s$	$0,03s$	$1,20s$

TABLE 2 – Temps de calcul

- **Tritype**. Ce programme est un programme classique qui a été utilisé très souvent en test et vérification de programmes. Il prend en entrée trois entiers (les côtés d'un triangle) et retourne 3 si les entrées correspondent à un triangle équilatéral, 2 si elles correspondent à un triangle isocèle, 1 si elles correspondent à un autre type de triangle, 4 si elles ne correspondent pas à un triangle valide.
- **TriPerimetre**. Ce programme a exactement la même structure de contrôle que tritype. La différence est que TriPerimetre renvoie la somme des côtés du triangle si les entrées correspondent à un triangle valide, et -1 dans le cas inverse.

Pour chacun de ces programmes, nous avons considéré différentes versions érronées.

Nous avons aussi évalué notre approche sur les programmes TCAS (Traffic Collision Avoidance System) de la suite de test Siemens[26]. Il s'agit là aussi d'un benchmark bien connu qui correspond à un système d'alerte de trafic et d'évitement de collisions aériennes. Il y a 41 versions erronées et 1608 cas de tests. Nous avons utilisé toutes les versions erronées sauf celles dont l'indice $AltLayerValue$ déborde du tableau $PositiveRAAltThresh$ car les débordements de tableau ne sont pas traités dans `CPBPV`. A savoir, les versions **TcasKO...TcasKO41**. Les erreurs dans ces programmes sont provoquées dans des endroits différents. 1608 cas de tests sont proposés, chacun correspondant à contre-exemple. Pour chacun de ces cas de test T_j, on construit un programme $TcasV_iT_j$ qui prend comme entrée le contre-exemple, et dont postcondition correspond à la sortie correcte attendue.

Le code source de l'ensemble des programmes est disponible à l'adresse `http://www.i3s.unice.fr/~bekkouch/Bench_Mohammed.html`.

La table 1 contient les résultats pour le premier ensemble de programmes :

- Pour `LocFaults` nous affichons la liste des MCS. La première ligne correspond aux MCS identifiés sur le chemin initial. Les lignes suivantes aux MCS identifiés sur les chemins pour lesquels la postcondition est satisfaite lorsqu'une condition est déviée. Le numéro de la ligne correspondant à la condition est souligné.
- Pour `BugAssist` les résultats correspondent à la fusion de l'ensemble des compléments des MSS calculés, fusion qui est opérée par `BugAssist` avant l'affichage des résultats.

Sur ces benchmarks les résultats de `LocFaults` sont plus concis et plus précis que ceux de `BugAssist`.

La table 2 fournit les temps de calcul : dans les deux cas, P correspond au temps de prétraitement et L au temps de calcul des MCS. Pour `LocFaults`, le temps de pré-traitement inclut la traduction du programme Java en un arbre de syntaxe abstraite avec l'outil JDT (Eclipse Java development tools), ainsi que la construction du CFG dont les noeuds sont des ensembles de contraintes. C'est la traduction Java qui est la plus longue. Pour `BugAssist`, le temps de prétraitement est celui la construction de la formule SAT. Globalement, les performances de `LocFaults` et `BugAssist` sont similaires bien que le processus d'évaluation de nos systèmes de contraintes soit loin d'être optimisé.

La table 3 donne les résultats pour les programmes de la suite `TCAS`. La colonne Nb_E indique pour chaque programme le nombre d'erreurs qui ont été introduites dans le programme alors que la colonne Nb_CE donne le nombre de contre-exemples. Les colonnes LF et BA indiquent respectivement le nombre de contre-exemples pour lesquels `LocFaults` et `BugAssist` ont identifié l'instruction erronée. On remarquera que `LocFaults` se compare favorablement à `BugAssist` sur ce benchmark qui ne contient quasiment aucune instruction arithmétique ; comme précédemment le nombre d'instructions suspectes identifiées par `Loc-`

Programme	Nb_E	Nb_CE	LF	BA
TcasKO	1	131	131	131
TcasKO2	2	67	67	67
TcasKO3	1	23	2	23
TcasKO4	1	20	16	20
TcasKO5	1	10	10	10
TcasKO6	3	12	36	24
TcasKO7	1	36	23	0
TcasKO8	1	1	1	0
TcasKO9	1	7	7	7
TcasKO10	6	14	16	84
TcasKO11	6	14	16	46
TcasKO12	1	70	52	70
TcasKO13	1	4	3	4
TcasKO14	1	50	6	50
TcasKO16	1	70	22	0
TcasKO17	1	35	22	0
TcasKO18	1	29	21	0
TcasKO19	1	19	13	0
TcasKO20	1	18	18	18
TcasKO21	1	16	16	16
TcasKO22	1	11	11	11
TcasKO23	1	41	41	41
TcasKO24	1	7	7	7
TcasKO25	1	3	0	3
TcasKO26	1	11	11	11
TcasKO27	1	10	10	10
TcasKO28	2	75	74	121
TcasKO29	2	18	17	0
TcasKO30	2	57	57	0
TcasKO34	1	77	77	77
TcasKO35	4	75	74	115
TcasKO36	1	122	120	0
TcasKO37	4	94	110	236
TcasKO39	1	3	0	3
TcasKO40	2	122	0	120
TcasKO41	1	20	17	20

TABLE 3 – Nombre d'erreurs localisés pour TCAS

`Faults` est dans l'ensemble nettement inférieur à celui de `BugAssist`.

Les temps de calcul de `BugAssist` et `LocFaults` sont très similaires et inférieurs à une seconde pour chacun des benchmarks de la suite `TCAS`.

5 Discussion

Nous avons présenté dans cet article une nouvelle approche pour l'aide à la localisation d'erreurs qui utilise quelques spécificités de la programmation par contraintes. Les premiers résultats sont encourageants mais doivent encore être confirmés sur des programmes plus importants et contenant plus d'opérations arithmétiques.

Au niveau des résultats obtenus `LocFaults` est plus précis que `BugAssist` lorsque les erreurs sont sur le chemin du contre exemple ou dans une des conditions du chemin du contre-exemple. Ceci provient du fait que `BugAssist` et `LocFaults` ne calculent pas exactement la même chose :

`BugAssist` calcule les compléments des différents sous-ensembles obtenus par `MaxSat,` c'est à dire des sous-ensembles de clauses satisfiables de cardinalité maximale. Certaines "erreurs " du programme ne vont pas être identifiées par `BugAssist` car les contraintes correspondantes ne figurent pas dans le complément d'un sous ensemble de clauses satisfiables de cardinalité maximale.

`LocFaults` calcule des MCS, c'est à dire le complément d'un sous-ensemble de clauses maximal, c'est à dire auquel on ne peut pas ajouter d'autre clause sans le rendre inconsistant, mais qui n'est pas nécessairement de cardinalité maximale.

`BugAssist` identifie des instructions suspectes dans l'ensemble du programme alors que `LocFaults` recherche les instructions suspectes sur un seul chemin.

Les travaux futurs concernent à la fois une réflexion sur le traitement des boucles (dans un cadre de bounded-model checking) et l'optimisation de la résolution pour les contraintes numériques. On utilisera aussi les MCS pour calculer d'autres informations, comme le MUS qui apportent une information complémentaire à l'utilisateur.

Remerciements :

Nous tenons à remercier Hiroshi Hosobe, Yahia Lebbah, Si-Mohamed Lamraoui et Shin Nakajima pour les échanges fructueux que nous avons eus. Nous tenons aussi à remercier Olivier Ponsini pour son aide et ses précieux conseils lors de la réalisation du prototype de notre système.

Références

[1] Thomas Ball, Mayur Naik, and Sriram K. Rajamani. From symptom to cause : localizing errors in counterexample traces. In Proceedings of POPL, pages 97–105. ACM, 2003.

[2] Michael Barnett and K. Rustan M. Leino. Weakest-precondition of unstructured programs. In PASTE'05, ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering, pages 82–87. ACM, 2005.

[3] Ilan Beer, Shoham Ben-David, Hana Chockler, Avigail Orni, and Richard J. Trefler. Explaining counterexamples using causality. In Proceedings of CAV, volume 5643 of Lecture Notes in Computer Science, pages 94–108. Springer, 2009.

[4] Elazar Birnbaum and Eliezer L. Lozinskii. Consistent subsets of inconsistent systems : structure and behaviour. J. Exp. Theor. Artif. Intell., 15(1) :25–46, 2003.

[5] John W. Chinneck. Localizing and diagnosing infeasibilities in networks. INFORMS Journal on Computing, 8(1) :55–62, 1996.

[6] John W. Chinneck. Fast heuristics for the maximum feasible subsystem problem. INFORMS Journal on Computing, 13(3) :210–223, 2001.

[7] John W. Chinneck. Feasibility and Infeasibility in Optimization : Algorithms and Computational Methods. Springer, 2008.

[8] Hélène Collavizza, Michel Rueher, and Pascal Van Hentenryck. Cpbpv : a constraint-programming framework for bounded program verification. Constraints, 15(2) :238–264, 2010.

[9] Hélène Collavizza, Nguyen Le Vinh, Olivier Ponsini, Michel Rueher, and Antoine Rollet.

Constraint-based bmc : a backjumping strategy. STTT, 16(1) :103–121, 2014.

[10] Alexander Felfernig, Monika Schubert, and Christoph Zehentner. An efficient diagnosis algorithm for inconsistent constraint sets. AI EDAM, 26(1) :53–62, 2012.

[11] Andreas Griesmayer, Roderick Bloem, and Byron Cook. Repair of boolean programs with an application to c. In Proceedings of CAV, volume 4144 of Lecture Notes in Computer Science, pages 358–371. Springer, 2006.

[12] Andreas Griesmayer, Stefan Staber, and Roderick Bloem. Automated fault localization for c programs. Electr. Notes Theor. Comput. Sci., 174(4) :95–111, 2007.

[13] Alex Groce, Sagar Chaki, Daniel Kroening, and Ofer Strichman. Error explanation with distance metrics. STTT, 8(3) :229–247, 2006.

[14] Alex Groce, Daniel Kroening, and Flavio Lerda. Understanding counterexamples with explain. In Proceedings of CAV, volume 3114 of Lecture Notes in Computer Science, pages 453–456. Springer, 2004.

[15] James A. Jones and Mary Jean Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In ASE, IEEE/ACM International Conference on Automated Software Engineering, pages 273–282. ACM, 2005.

[16] James A. Jones, Mary Jean Harrold, and John T. Stasko. Visualization of test information to assist fault localization. In ICSE,Proceedings of the 22rd International Conference on Software Engineering, pages 467–477. ACM, 2002.

[17] Manu Jose and Rupak Majumdar. Bug-assist : Assisting fault localization in ansi-c programs. In Proceedings of CAV, volume 6806 of Lecture Notes in Computer Science, pages 504–509. Springer, 2011.

[18] Manu Jose and Rupak Majumdar. Cause clue clauses : error localization using maximum satisfiability. In Proceedings of PLDI, pages 437–446. ACM, 2011.

[19] Ulrich Junker. Quickxplain : Preferred explanations and relaxations for over-constrained problems. In Proceedings of AAAI, pages 167–172. AAAI Press / The MIT Press, 2004.

[20] Si-Mohamed Lamraoui and Shin Nakajima. A formula-based approach for automatic fault localization in imperative programs. NII research report, Submitted For publication, 6 pages, February, 2014.

[21] Mark H. Liffiton and Ammar Malik. Enumerating infeasibility : Finding multiple muses quickly. In Proc. of CPAIOR, volume 7874 of Lecture Notes in Computer Science, pages 160–175. Springer, 2013.

[22] Mark H. Liffiton and Karem A. Sakallah. Algorithms for computing minimal unsatisfiable subsets of constraints. J. Autom. Reasoning, 40(1) :1–33, 2008.

[23] Yongmei Liu and Bing Li. Automated program debugging via multiple predicate switching. In Proceedings of AAAI. AAAI Press, 2010.

[24] Joao Marques-Silva, Federico Heras, Mikolás Janota, Alessandro Previti, and Anton Belov. On computing minimal correction subsets. In Proc. of IJCAI. IJCAI/AAAI, 2013.

[25] Manos Renieris and Steven P. Reiss. Fault localization with nearest neighbor queries. In Proceedings of ASE, pages 30–39. IEEE Computer Society, 2003.

[26] David S. Rosenblum and Elaine J. Weyuker. Lessons learned from a regression testing case study. Empirical Software Engineering, 2(2) :188–191, 1997.

[27] Mehrdad Tamiz, Simon J. Mardle, and Dylan F. Jones. Detecting iis in infeasible linear programmes using techniques from goal programming. Computers & OR, 23(2) :113–119, 1996.

[28] Xiangyu Zhang, Neelam Gupta, and Rajiv Gupta. Locating faults through automated predicate switching. In Proceedings of ICSE, pages 272–281. ACM, 2006.

Comparaison de BTD avec des stratégies d'exploration "intelligentes" pour une sélection automatique d'algorithmes .

L. Blet[1,3] **S. N. Ndiaye**[1,2] **C. Solnon**[1,3]

[1] Université de Lyon - LIRIS
[2] Université Lyon 1, LIRIS, UMR5205, F-69622 France
[3] INSA-Lyon, LIRIS, UMR5205, F-69621, France
{loic.blet,samba-ndojh.ndiaye,christine.solnon}@liris.cnrs.fr

Abstract

Nous considérons un solveur générique de problèmes de satisfaction de contraintes (CSP) binaires, paramétré par des choix de haut niveau, à savoir le type de recherche, le niveau de propagation de contraintes et l'heuristique de choix de variables. Nous comparons expérimentalement 18 configurations de ce solveur générique sur plus d'un millier d'instances. Un premier but est de comprendre la complémentarité des différents types de recherche, avec une attention particulière sur Backtracking with Tree-Decomposition (BTD). Le second objectif est d'introduire un sélecteur d'algorithme par instance pour choisir automatiquement la meilleure approche pour chaque nouvelle instance à résoudre. Nous montrons expérimentalement que ce sélecteur obtient de meilleures performances par rapport à nos 18 configurations initiales.

1 Introduction

Les méthodes complètes basées sur la technique du backtracking résolvent les CSP en construisant un arbre de recherche. Pour le BackTracking Chronologique (CBT) [27], cet arbre est exploré en profondeur d'abord. Lors d'un échec, la recherche revient au dernier point de choix. Cela mène à une exploration inutile de sous-arbres quand l'échec n'est pas dû à la dernière décision. Pour surmonter ce problème, des techniques de backtracking intelligent ont été proposées, comme Conflict-directed Back-Jumping (CBJ) [26] et Dynamic BackTracking (DBT) [12]. Elles exploitent dynamiquement la structure du problème pour retourner à la cause de l'échec et ainsi éviter une exploration redondante de certaines zones de l'espace de recherche. Backtracking with Tree Decomposition (BTD) [18] capture la structure statique du problème en identifiant des sous-problèmes indépendants.

CBJ, DBT et CBT ont déjà été comparés expérimentalement (voir par exemple, [21]). BTD a aussi été comparé à CBT et CBJ (voir, [18]). Cependant, à notre connaissance, c'est la première fois que BTD est comparé à CBJ et DBT sur un ensemble de problèmes aussi conséquent. Il est intéressant de les comparer car ils exploitent tous la structure pour guider la recherche. CBJ et DBT exploitent une structure dynamique grâce aux explications des incohérences, tandis que BTD exploite une structure statique grâce à une décomposition arborescente. De plus, ces différents mécanismes de backtracking peuvent être combinés avec différents niveaux de propagation de contraintes, comme Forward-Checking (FC) et Maintaining Arc Consistency (MAC). Ils peuvent également être combinés avec différentes heuristiques de choix de variables. En particulier, [8] exploite les échecs rencontrés durant la recherche pour mieux choisir la prochaine variable à affecter. Ainsi, cette heuristique exploite aussi la structure de l'instance pour guider la recherche.

Dans cet article, nous décrivons un solveur de CSP générique qui étend celui de [21]. Il a trois paramètres : (i) la stratégie de recherche – CBT, CBJ, DBT ou BTD ; (ii) Le niveau de propagation de contraintes – FC ou MAC ; (iii) l'heuristique de choix de variables – minDomaine sur degré dynamique ou degré dynamique pondéré.

Une première contribution de l'article est de comparer expérimentalement différentes configurations de ce solveur générique sur plus de mille instances. En particulier, nous comparons 4 configurations basées sur BTD avec d'autres variantes basées sur des techniques de backtracking in-

telligent. Cette vaste étude expérimentale nous montre que même si deux configurations ont des taux de succès globaux meilleurs que toutes les autres, certaines configurations (notamment basées sur BTD) ayant un taux de succès global faible, se comportent très bien sur un grand nombre d'instances. Plus particulièrement, nous identifions 13 configurations complémentaires telles que chaque instance est bien résolue par au moins une de ces configurations.

Une deuxième contribution est d'introduire un sélecteur d'algorithme par instance pour notre solveur générique. Cela peut s'avérer plus efficace que la combinaison de différentes techniques visant la définition d'une méthode performante sur tous les types d'instances [7]. À l'image de méthodes récentes (voir,[25, 19, 1]), nous extrayons des descripteurs pour nos instances et utilisons des techniques d'apprentissage pour construire un modèle de sélection. Le choix des configurations parmi lesquelles le sélecteur va opérer est un point crucial : le but est de conserver un sous-ensemble de configurations S avec des performances complémentaires telles que S contienne une configuration efficace pour chaque instance dans l'ensemble d'apprentissage. Nous exprimons ce problème comme un problème de couverture d'ensemble et utilisons un algorithme glouton pour déterminer une solution approchée.

L'article est organisé comme suit. En section 2 nous décrivons notre cadre générique pour résoudre les CSPs. En section 3, nous testons différentes configurations de ce cadre. En section 4, nous décrivons notre sélecteur d'algorithme par instance. En section 5 nous comparons notre sélecteur aux meilleures configurations et à un meilleur solveur virtuel. Nous concluons en section 6 avec des pistes futures.

2 Cadre générique pour les CSPs binaires

Contexte. Un CSP est défini par un triplet (X, D, C). X est un ensemble fini de variables. D associe un ensemble fini de valeurs $D(x_i)$ à chaque variable $x_i \in X$. C est un ensemble de contraintes. Chaque contrainte est définie sur un sous-ensemble de variables et définit des tuples de valeurs pouvant être affectées simultanément à ces variables. Une contrainte binaire porte sur deux variables. Dans cet article nous considérons des CSPs binaires qui ne contiennent que des contraintes binaires. Une solution est une affectation de toutes les variables satisfaisant toutes les contraintes.

Dans cet article, l'accent est mis sur les méthodes complètes basées sur la technique du backtracking qui structurent l'espace de recherche en arbre où les nœuds correspondent à une affectation d'une valeur à une variable. Nous introduisons un algorithme générique qui est paramétré par le mécanisme de backtracking, le niveau de propagation de contraintes et l'heuristique de choix de variables. Cet algorithme générique est une extension de celui décrit dans [21] et nous permet de comparer, dans un cadre unifié, des approches de l'état de l'art pour les CSPs binaires.

Backtracking. Dans le Backtracking Chronologique (CBT) [27], l'arbre est exploré en profondeur d'abord. Lors d'un échec, la recherche revient sur le dernier nœud. Si la cause n'est pas due à la dernière décision, CBT explore inutilement des sous-arbres.

Conflict directed BackJumping (CBJ) [26] retourne immédiatement à la dernière variable affectée en cause dans l'échec et désaffecte toute variable affecté après elle. Dans cette étude nous utilisons des techniques de [12, 2, 17] pour obtenir une version de CBJ semblable à celle de [21]. Elle maintient pour chaque valeur testée sans succès l'ensemble des variables affectées en cause dans cet échec. Si cet ensemble est vide la valeur est définitivement supprimée.

Dynamic BackTracking (DBT) [12] a pour objectif d'améliorer CBJ en ne désaffectant pas les variables entre la variable courante et celle en cause dans l'échec vu qu'elles ne sont pas responsables de cet échec. À cause des mauvaises performances de DBT combiné à FC et une bonne heuristique de choix de variables (voir, [3]), [28] propose une nouvelle version de CBJ en réordonnant rétroactivement les variables affectées (CBJR). Après chaque affectation $x \leftarrow v$, le critère utilisé par l'heuristique de choix de variables est recalculé en tenant compte de l'état actuel du problème pour éventuellement remonter la variable x dans l'arbre de recherche (par exemple si beaucoup de valeurs ont été supprimées du domaine de la variable, elle peut être replacée avant d'autres variables avec un domaine plus large). Pour conserver la complétude, x ne peut pas être placée au-dessus d'une variable participant au filtrage ou à l'échec d'une valeur dans son domaine.

Enfin, Backtracking with Tree Decomposition (BTD) [18] utilise une décomposition arborescente du graphe des contraintes qui capture la structure du problème en identifiant des sous-problèmes indépendants. BTD calcule l'ordre dans lequel les sous-problèmes doivent être résolus, donnant un ordre partiel sur les variables. De plus, il enregistre des *goods* associés aux solutions des sous-problèmes et des *nogoods* associés aux échecs des sous-problèmes. Ces informations évitent de résoudre un même sous-problème plus d'une fois. Dans cette étude, la décomposition arborescente est calculée en utilisant l'heuristique minimum-fill [20] pour trianguler les graphes des contraintes.

Propagation de contraintes. À chaque nœud de l'arbre de recherche, les contraintes sont propagées pour filtrer certaines valeurs du domaine des variables non affectées et pour détecter des incohérences locales. Nous considérons deux mécanismes [27] : Forward Checking (FC) qui supprime les valeurs non arc cohérentes avec la

dernière affectation ; Maintaining Arc Consistency (MAC) qui assure l'arc cohérence de toutes les contraintes.

Heuristique de choix de variables. À chaque nœud de l'arbre, la recherche choisit la prochaine variable à affecter parmi celles non affectées. Une heuristique de choix de variables classique est minDomaine, qui choisit une variable qui a le plus petit domaine. Dans cette étude, nous en considérons deux améliorations connues [5, 8] :

- minDomaine sur degré dynamique (d) choisit une variable x minimisant le ratio entre la taille de $D(x)$ et le nombre de variables non affectées partageant une contrainte avec x,
- minDomaine sur degré dynamique pondéré (w) choisit une variable x minimisant le ratio entre la taille de $D(x)$ et la somme des poids des contraintes incluant x avec d'autres variables non affectées, où le poids d'une contrainte est le nombre d'échecs qu'elle a causée depuis le début de la recherche.

Avec BTD, l'heuristique de choix de variables est limitée au cluster actuel de la décomposition (voir [16]).

Cadre générique. [21] définit un premier cadre générique qui englobe plusieurs méthodes de backtracking de l'état de l'art. Nous étendons ce cadre avec deux nouveaux mécanismes de backtracking, CBJR et BTD. De ce cadre générique, nous pouvons obtenir des configurations notées par des triplets (b, c, o) où $b \in$ {CBT, CBJ, DBT, CBJR, BTD} définit le mécanisme de backtracking, $c \in$ {FC,MAC} définit le niveau de propagation de contraintes et $o \in$ {d,w} définit l'heuristique de choix de variables. Nous avons 18 configurations – et non 20 – car CBJR est combiné uniquement avec FC comme dans [28]. Sa combinaison avec MAC sera pour des travaux futurs.

Dans les configurations (CBT,MAC,*) et (BTD,MAC,*), nous utilisons AC2001 [6] pour maintenir la cohérence d'arc, tandis que dans (CBJ,MAC,*) et (DBT,MAC,*) nous utilisons AC3 [23]. AC2001 considère un ordre sur les valeurs dans les domaines et enregistre pour chaque valeur sa première valeur compatible dans les autres domaines. Si cette valeur est supprimée, il cherche une nouvelle valeur compatible en partant de la position de la valeur supprimée. AC3 est différent sur ce point car il parcourt à nouveau tout le domaine depuis le début pour chercher une nouvelle valeur compatible.

Toutes les configurations sont non déterministes : l'heuristique de choix de variables départage les égalités aléatoirement et nous ne considérons pas d'ordre de choix de valeurs, qui sont choisies aléatoirement.

Enfin, pour toutes les configurations, nous commençons par décomposer le graphe de contraintes en composantes connexes pour obtenir des sous-problèmes indépendants résolus séparément. Aussi, chaque sous-problème est rendu arc cohérent avant de commencer sa résolution.

3 Comparaison expérimentale

3.1 Ensemble de problèmes considérés

Notre ensemble de problèmes est composé de 1092 instances regroupées en 6 classes décrites dans le tableau 1. Les 5 premières classes viennent de la compétition CSP'08 pour laquelle nous n'avons considéré que les instances binaires. Pour les classes contenant des instances similaires, nous n'avons gardé que les dix premières. Nous avons enlevées toutes les instances qui ne sont résolues par aucune de nos 18 configurations en 30 minutes (sur 15 essais).

La dernière classe (STRUCT) contient des instances structurées générées aléatoirement comme dans [15]. Ces instances ont une structure similaire à celle des instances RLFAP qui sont des problèmes réels. La structure est définie par un arbre de clusters de variables et le niveau de structuration dépend de la densité de contraintes dans les clusters et de la taille des clusters. Cette classe contient des sous-classes d'instances de différents niveaux de structuration, différentes tailles et différentes duretés de contraintes.

3.2 Résultats expérimentaux

Le tableau 2 compare les taux de succès de 18 configurations à différents temps CPU sur un Intel(R) Xeon(R) CPU E5-2670 0 @ 2.60GHz, 20480 KB cache size, 3GB RAM. Toutes les configurations étant non déterministes, nous les avons lancés 15 fois pour chaque instance.

(CBT,MAC,w) est la meilleure configuration avec un temps CPU limité à 1000 secondes. Cela n'est pas surprenant et a déjà été observé dans [21]. Cependant, (CBJ,MAC,w) prend la tête lorsque le temps CPU limite dépasse 1000 secondes. Sans surprise, nous remarquons que les configurations utilisant l'heuristique w pour ordonner les variables font mieux que les configurations utilisant d (voir, [8]). Le gain dépend cependant de la méthode de backtracking considérée comme montré dans [9]. En particulier, utiliser w améliore considérablement les performances de DBT et CBT, mais plus modestement celles de CBJ. Par ailleurs, les configurations utilisant DBT et FC obtiennent de mauvais résultats.

Ces taux de succès sur les 1092 problèmes cachent des réalités très différentes en regardant chaque instance. Certaines configurations ayant des taux de succès assez bas sur l'ensemble des problèmes sont les meilleures sur certaines instances. Nous appliquons une règle simple pour décider si une configuration est la *meilleure* pour une instance : nous comparons en premier le nombre d'essais réussis sur 15 en moins de 30 minutes et nous départageons les égalités par le temps moyen de résolution.

Classe	#Instances	#Variables			#Valeurs			#Contraintes			Dureté des contraintes		
		min	moy	max	min	moy	max	min	moy	max	min	moy	max
ACAD	75	10	116	500	2	146	2187	45	691	4950	0.001	0.692	0.998
PATT	238	16	263	1916	3	66	378	48	4492	65390	0.002	0.795	0.996
QRND	80	50	220	315	7	11	20	451	2968	4388	0.122	0.578	0.823
RAND	206	23	37	59	8	36	180	84	282	753	0.095	0.613	0.984
REAL	193	200	628	1000	2	152	802	1235	6394	17447	0.0	0.519	1.0
STRUCT	300	150	257	500	20	23	25	617	1641	3592	0.544	0.647	0.753

Table 1: Classes de l'ensemble de problèmes. Pour chaque classe, le tableau présente son nom, le nombre d'instances, le nombre de variables, la taille des domaines, le nombre de contraintes et la dureté des contraintes (minimum, moyenne et valeurs maximum).

t	CBT				DBT				CBJ				CBJR		BTD			
	FC		MAC		FC		MAC		FC		MAC		FC		FC		MAC	
	d	w	d	w	d	w	d	w	d	w	d	w	d	w	d	w	d	w
1	37.0	41.8	43.0	**47.1**	33.8	37.7	35.8	37.9	41.3	39.6	38.0	39.7	39.9	39.1	31.2	31.6	31.1	35.1
10	47.6	56.8	55.9	**67.5**	38.8	50.5	49.4	54.1	55.2	55.0	54.3	57.6	53.3	55.0	51.3	52.2	51.0	58.6
100	55.3	69.4	68.2	**84.4**	41.5	66.5	60.0	74.5	70.5	72.6	74.2	79.6	66.9	72.6	68.6	72.6	71.5	79.6
1000	61.1	81.5	75.8	**93.5**	45.8	80.3	68.0	89.5	85.6	88.1	88.8	**93.5**	78.1	88.3	77.2	83.1	81.5	90.2
1800	61.7	83.2	77.0	94.6	46.7	83.7	69.3	91.8	88.0	91.0	90.4	**95.1**	79.5	90.9	77.8	84.0	82.7	91.2

Table 2: Comparaison des taux de succès globaux. Pour chaque limite de temps CPU t (en secondes), le tableau donne le pourcentage d'essais réussis de chaque configuration au temps t sur 15 essais et 1092 instances.

La première ligne du tableau 3 affiche le pourcentage d'instances pour lequel une configuration est la meilleure. Cela montre que même si (CBT,FC,d) ne résout que 61.7% d'instances après 30 minutes, c'est la meilleure configuration pour 30.6% des 1092 instances. Il est connu que des configurations simples comme (CBT,FC,d) peuvent avoir de meilleures performances que des configurations plus évoluées sur des instances faciles, pour lesquelles il n'y a pas besoin de mécanisme certes intelligent mais coûteux, tandis qu'elles ont habituellement de très mauvaises performances sur des instances plus dures. Dans la deuxième ligne du tableau 3, nous avons enlevé les instances faciles : nous considérons qu'une instance est facile si elle est résolue par (CBT,MAC,w) en moins d'une seconde sur les 15 essais. Avec cette définition, 470 instances sont faciles et 622 plus difficiles. Le tableau 3 nous montre que certaines configurations (comme celles utilisant DBT) sont meilleures sur très peu d'instances difficiles.

Comme beaucoup de configurations peuvent avoir des résultats proches pour une instance donnée, nous étudions le nombre d'instances pour lesquelles une configuration est bonne : nous considérons qu'une configuration est *bonne* pour une instance si c'est la meilleure ou s'il n'y a pas un écart statistiquement significatif entre ses résultats et ceux de la meilleure. Nous utilisons le test de Student avec $p = 0,01$ pour décider si une configuration est significativement meilleure qu'une autre sur une instance. La troisième ligne du tableau 3 montre le pourcentage d'instances difficiles pour lesquelles une configuration est bonne. Encore une fois, nous observons que les configurations qui sont bonnes sur beaucoup d'instances n'ont pas toujours un taux de succès global élevé. En particulier, (BTD,FC,d) et (CBT,FC,d) qui sont bonnes sur le plus grand nombre d'instances sont loin d'avoir les meilleurs taux de succès globaux dans le tableau 2.

Toutes les configurations sont bonnes pour au moins une instance. Il arrive qu'une configuration ne soit bonne que sur des instances ou d'autres le sont aussi : elle est dominée. La quatrième ligne du tableau 3 affiche le pourcentage d'instances difficiles pour lesquelles une configuration est la meilleure et toutes les autres sont significativement moins bonnes. 6 configurations sont dominées : (DBT,FC,w), (DBT,MAC,*), (CBJ,FC,*) et (CBJR,FC,d). Donc nous avons enlevé ces configurations de notre étude, sauf (CBJR,FC,d) : il y a 10 instances pour lesquelles toutes les bonnes configurations sont parmi les 6 dominées, puisque (CBJR,FC,d) est bonne sur ces 10 instances, nous l'avons gardée. La dernière ligne du tableau 3 donne le pourcentage d'instances difficiles pour lesquelles une configuration est significativement meilleure que les autres à l'exception de (DBT,FC,w), (DBT,MAC,*) et (CBJ,FC,*). Cela nous montre que les 13 configurations restantes sont complémentaires : chacune est meilleure sur certaines instances.

Le tableau 3 montre que BTD est très efficace sur certaines instances. (BTD,FC,d) est significativement meilleure que les autres configurations sur plus de 15% des instances difficiles. Sur d'autres instances BTD est mauvais donc son taux de succès global est plutôt bas comparé à d'autres approches. Pour comprendre quelles in-

	CBT				DBT				CBJ				CBJR		BTD			
	FC		MAC		FC		MAC		FC		MAC		FC		FC		MAC	
	d	w	d	w	d	w	d	w	d	w	d	w	d	w	d	w	d	w
#meilleure / toutes	**30.6**	6.6	11.5	10.9	2.9	0.6	1.8	0.5	1.2	1.6	1.9	1.5	0.7	1.7	16.6	6.2	0.8	2.3
#meilleure / diff	21.2	4.3	8.5	14.0	1.3	0.3	1.6	0.2	1.1	0.3	0.6	2.1	0.8	2.1	**27.7**	10.1	1.4	3.9
#bonne / diff	31.5	13.7	12.4	21.7	2.7	6.9	4.5	4.0	5.3	4.7	4.5	8.7	3.2	5.5	**40.2**	17.2	5.3	9.3
#seule bonne/ diff	11.3	0.6	5.6	7.9	0.2	0	0	0	0	0	0.2	0.5	0	1.6	**14.5**	3.7	0.2	1.8
#seule bonne / diff	11.9	1.0	5.8	8.5	0.3	–	–	–	–	–	0.3	1.0	1.6	1.8	**15.1**	3.7	0.3	1.8

Table 3: Comparaison des pourcentages d'instances mieux résolues. Pour chaque configuration c la première ligne donne le pourcentage d'instance pour lequel c est la meilleure (sur 1092 instances) ; la deuxième ligne (respectivement la troisième) donne le pourcentage d'instances difficiles (parmi 622 instances difficiles) pour lequel c est la meilleure (respectivement c n'est pas significativement moins bonne que la meilleure) ; la quatrième ligne (respectivement la cinquième) donne le pourcentage d'instances difficiles pour lequel c est la meilleure et toutes les autres configurations sont significativement moins bonnes que c (respectivement toutes les autres configurations sauf (DBT,FC,w), (DBT,MAC,*) et (CBJ,FC,*) sont significativement moins bonnes que c).

	Nombre d'instances							Largeur d'arbre	Taille sep
	STRUCT	ACAD	PATT	REAL	QRND	RAND	Total	(moy)	(moy)
Décompose	180	0	10	5	0	0	195	11.3%	3.0%
Décompose pas	63	5	63	75	15	69	290	37.5%	29.5%
Agnostique	26	16	20	5	3	67	137	56.7%	25.7%

Table 4: Description des 3 ensembles d'instances. Pour chaque ensemble, le tableau affiche le nombre d'instances de chaque classe, la largeur arborescente moyenne et la taille moyenne des séparateurs (en pourcentage du nombre de variables) de la décomposition arborescente.

stances sont mieux résolues par BTD, nous partitionnons les 622 instances difficiles en trois ensembles : l'ensemble *Décompose* contient les instances difficiles mieux résolues uniquement par (BTD,*,*) et aucune configuration non-BTD n'y est bonne ; l'ensemble *Décompose pas* contient les instances difficiles mieux résolues par une des configurations non-BTD et aucune des (BTD,*,*) n'y est bonne ; l'ensemble *Agnostique* pour le reste des instances.

Le tableau 4 nous montre comment les instances sont réparties dans ces ensembles. Presque toutes les instances de l'ensemble *Décompose* viennent de la classe STRUCT, qui contient des instances structurées. Ce n'est pas une surprise que les variantes de BTD aient de meilleures performances que les variantes de CBT sur ces instances (voir [18]). Comme BTD n'a pas encore été comparé avec des mécanismes de backtracking intelligents, remarquons que BTD est meilleur qu'eux sur ces instances. Seules 15 instances de la compétition CSP'08 appartiennent à l'ensemble *Décompose* : la plupart des instances a été générée aléatoirement et ne possède pas de structure statique exploitable. En regardant les paramètres de la décomposition arborescente, nous remarquons que les instances de l'ensemble *Décompose* ont une largeur arborescente plus petite (trois fois plus petite que dans l'ensemble *Décompose pas*) et une taille de séparateur plus petite (dix fois plus petite que dans l'ensemble *Décompose pas*). Les instances de l'ensemble *Agnostique* ont une grande largeur arborescente. En fait, quand la largeur arborescente est proche du nombre de variables, BTD se comporte comme CBT vu que presque toutes les variables sont dans le même cluster.

4 Sélecteur d'algorithme par instance

Les résultats rapportés dans la section précédente montrent que 13 des 18 configurations sont complémentaires. De plus, les meilleures configurations sur certaines instances sont très mauvaises sur d'autres de telle sorte qu'elles n'ont pas le meilleur taux de succès moyen sur l'ensemble des problèmes. Cette illustration du théorème *no free lunch* motive notre recherche d'un sélecteur d'algorithme par instance visant à sélectionner une bonne configuration pour chaque nouvelle instance à résoudre. Dans cette première étude, nous n'essayons pas d'améliorer l'état de l'art des sélecteurs d'algorithme par instance tels que CPHydra [25], ISAC (Instance Specific Algorithm Configuration) [19] ou EISAC [24] (voir [1] pour une comparaison).

Notre but est de montrer qu'on peut utiliser une bibliothèque d'apprentissage prête à l'emploi pour construire rapidement un sélecteur d'algorithme efficace exploitant la complémentarité de plusieurs solveurs. C'est une première étape d'une étude plus large et notre sélecteur pourrait être étendu en un ordonnanceur sur un sous-ensemble de solveurs de façon assez directe.

L'idée est d'entraîner un classifieur supervisé [4] en lui donnant un ensemble d'apprentissage d'instances CSPs : chaque instance exemple est décrite par des descripteurs

et une étiquette correspondant à la meilleure configuration pour cette instance. L'étape d'entraînement permet au classifieur d'apprendre un modèle qui sera utilisé pour choisir une configuration pour de nouvelles instances à résoudre.

Dans cette section, nous listons d'abord les descripteurs utilisés pour caractériser les CSPs (section 4.1). Puis nous expliquons comment choisir un sous-ensemble de configurations utilisées comme ensemble d'étiquettes (section 4.2). Enfin, nous montrons comment apprendre un modèle à partir d'un ensemble d'apprentissage (section 4.3).

4.1 Description des instances par descripteurs

Nous décrivons chaque CSP par un ensemble de descripteurs. Le but est de trouver un ensemble pertinent de descripteurs qui caractérise adéquatement les instances de telle sorte que le classifieur peut les utiliser pour prédire l'étiquette. Ces descripteurs doivent être simple à extraire vu que le temps pour récupérer les descripteurs est inclus dans le temps final de résolution. Nous extrayons deux types de descripteurs : les descripteurs statiques extraits de l'instance et les descripteurs dynamiques extraits d'un court essai d'une configuration sur l'instance.

Descripteurs statiques. Nous extrayons les descripteurs suivants de chaque instance : nombre de variables, nombre de contraintes, taille des domaines (moyenne et écart-type), dureté des contraintes (à savoir le ratio du nombre de tuples interdits sur le nombre de tuples possibles (moyenne et écart-type)) et le degré des variables dans le graphe des contraintes (moyenne et écart-type).

Comme certaines instances ont plus d'une composante connexe dans leur graphe de contraintes, nous extrayons aussi les descripteurs suivants : nombre de composantes connexes, nombre de variables dans une composante connexe (moyenne et écart-type) et nombre de contraintes dans une composante connexe (moyenne et écart-type).

Enfin nous extrayons aussi des descripteurs d'une décomposition arborescente qui est calculée en utilisant l'algorithme minimum fill [20] pour trianguler le graphe des contraintes : nombre de clusters, taille maximum d'un séparateur et d'un cluster et la densité moyenne de contraintes dans un cluster (moyenne et écart-type).

Descripteurs dynamiques. Pour récupérer plus d'informations sur l'instance à résoudre, nous effectuons également un court essai pour en extraire des descripteurs. Nous limitons la tentative à 1 seconde. Les instances faciles (pour lesquelles il n'y a pas besoin de sélecteur) sont résolues durant cet essai. Pour les instances plus dures non résolues en 1 seconde, nous arrêtons l'essai à 1 seconde, récupérons des descripteurs dynamiques et statiques et donnons ces descripteurs au sélecteur qui choisit une configuration pour cette instance.

Comme (CBT,MAC,w) est la meilleure configuration à 1 seconde, nous la choisissons pour le court essai. De plus, cette configuration nous permet de récupérer des informations sur le poids des variables (donné par l'heuristique w) et le nombre de valeurs filtrées par MAC. Nous rassemblons les descripteurs dynamiques suivants : nombre de nœuds dans l'arbre de recherche, nombre d'échecs, nombre de valeurs supprimées par MAC (moyenne et écart-type) et poids d'une variable (moyenne et écart-type). Pour comprendre la dynamique d'un essai, nous rassemblons ces informations à 0,25, 0,5 et 1 seconde.

4.2 Sélection d'un sous-ensemble de configurations.

Pendant la phase d'entraînement, nous pourrions étiqueter chaque instance de l'ensemble d'entraînement avec la meilleure configuration (en moyenne sur 15 essais) parmi les 13 configurations restantes (après élimination des dominées (DBT,FC,w), (DBT,MAC,*) et (CBJ,FC,*)). Cependant, la classification devient plus difficile quand il y a plus de classes. Nous voulons étudier l'impact du nombre de classes sur la performance de notre sélecteur. Ainsi, nous avons testé différents sous-ensembles de configurations.

Plus précisément, nous choisissons un sous-ensemble S_k de k configurations où $2 \leq k \leq 13$ est un paramètre à fixer. Ces k configurations définissent les classes des instances d'entraînement : chaque instance i de l'ensemble d'entraînement est étiqueté avec une configuration de S_k qui est la meilleure pour i (en moyenne sur 15 essais). Dans notre cas, le but est de garder dans S_k des configurations qui ont des performances complémentaires de telle sorte que, pour chaque instance i de l'ensemble d'entraînement, il existe une configuration $c \in S_k$ bonne pour i.

En section 3, nous considérons qu'une configuration c est bonne pour une instance i si soit c'est la meilleure configuration pour i ou elle n'est pas significativement moins bonne que la meilleure. Nous gardons cette définition ici et choisissons les configurations de S_k en résolvant un problème de couverture : une instance i est *couverte* par une configuration c si c est bonne pour i ; le but est de trouver l'ensemble S_k contenant les k configurations maximisant le nombre d'instances couvertes dans l'ensemble d'entraînement. Ce problème étant NP-difficile, nous le résolvons approximativement de façon gloutonne : partant de la configuration S_1 qui couvre le plus d'instances nous définissons S_i de S_{i-1} en y ajoutant la configuration qui augmente le plus le nombre d'instances couvertes.

Le tableau 5 donne le pourcentage d'instances couvertes pour différents ensembles S_k sur les 622 instances difficiles. La configuration couvrant le plus d'instances est (BTD,FC,d) et S_1 ={(BTD,FC,d)} couvre environ 40% des instances. S_2 est obtenu en ajoutant (CBT,MAC,w) à S_1, S_2 couvre environ 60% des instances. Le tableau 5

Ensemble de configurations	% d'instances couvertes	Taux de succès d'un solveur virtuel idéal				
		1	10	100	1000	1800
S_1 ={(BTD,FC,d)}	40.2%	31.2%	51.3%	68.6%	77.2%	77.8%
$S_2 = S_1 \cup$ {(CBT,MAC,w)}	59.3%	52.6%	75.9%	91.2%	97.5%	97.9%
$S_3 = S_2 \cup$ {(CBT,FC,d)}	75.6%	54.4%	77.4%	91.9%	97.6%	98.0%
$S_4 = S_3 \cup$ {(CBT,MAC,d)}	82.2%	55.5%	77.6%	91.9%	97.6%	98.0%
$S_5 = S_4 \cup$ {(BTD,FC,w)}	88.1%	55.7%	78.1%	92.9%	98.4%	98.8%
$S_6 = S_5 \cup$ {(CBJ,MAC,w)}	91.2%	55.9%	78.2%	93.4%	99.0%	99.3%
$S_7 = S_6 \cup$ {(BTD,MAC,w)}	93.7%	55.9%	78.3%	93.4%	99.0%	99.4%
$S_8 = S_7 \cup$ {(CBJR,FC,w)}	96.1%	56.0%	78.5%	93.7%	99.2%	99.6%
$S_9 = S_8 \cup$ {(CBJR,FC,d)}	97.6%	56.3%	78.6%	93.8%	99.2%	99.6%
$S_{10} = S_9 \cup$ {(CBT,FC,w)}	98.9%	56.4%	79.0%	93.8%	99.2%	99.6%
$S_{11} = S_10 \cup$ {(BTD,MAC,d)}	99.4%	56.4%	79.0%	93.8%	99.2%	99.6%
$S_{12} = S_11 \cup$ {(DBT,FC,d)}	99.7%	56.4%	79.1%	93.8%	99.2%	99.6%
S_{13} = toutes configurations	100.0%	56.4%	79.1%	93.9%	99.2%	99.6%
B_1 = {(CBT,MAC,w)}	21.7%	47.1%	67.5%	84.4%	93.5%	94.6%
$B_2 = B_1 \cup$ {(CBJ,MAC,w)}	25.2%	47.3%	67.9%	86.2%	95.9%	96.9%
$B_3 = B_2 \cup$ {(DBT,MAC,w)}	25.9%	47.3%	68.0 %	86.2%	95.9%	97.0%

Table 5: Ensembles de configurations. Pour chaque ensemble, le tableau affiche le pourcentage d'instances difficiles couvertes par l'ensemble et le pourcentage d'essais réussis d'un solveur virtuel idéal qui choisit la meilleure configuration possible dans l'ensemble pour chaque instance, à différentes limites de temps (sur 15 essais et 1092 instances).

donne aussi le pourcentage d'essais réussis par le solveur virtuel idéal correspondant à différentes limites de temps. Le solveur virtuel idéal, noté VBS(S_k), choisit la meilleure configuration de S_k pour chaque instance à résoudre : un sélecteur basé sur S_k ne peut battre VBS(S_k). Enfin, le tableau 5 affiche les résultats obtenu par les meilleurs solveurs (B_1,B_2 et B_3) d'après les taux de succès globaux vus dans le tableau 2 : les taux de succès de VBS(B_2) et VBS(B_3) sont bien en dessous de ceux de VBS(S_2) et VBS(S_3), démontrant ainsi l'intérêt de choisir des solveurs complémentaires, plutôt que les meilleurs solveurs.

4.3 Entraînement du sélecteur.

Soient un ensemble S_k de configurations et un ensemble d'entraînement I d'instances tels que chaque instance $i \in I$ soit décrite par des descripteurs et une étiquette d'un solveur de S_k qui est le meilleur pour i, le but est d'entraîner un classifieur pour associer des instances avec des configurations. Cette classification supervisée (voir [4]) est résolue par de nombreuses approches. Nous utilisons la bibliothèque Weka [14, 13]. Nous avons comparé différents classifieurs supervisés. Les meilleurs résultats sont obtenus par ClassificationViaRegression [11] donc nous utilisons ce classifieur dans nos tests.

Une fois le classifieur entraîné, nous l'utilisons pour configurer notre solveur générique. Pour résoudre une instance i nous procédons de la sorte : d'abord nous lançons (CBT,MAC,w) sur i pendant 1 seconde ; si i n'est pas résolue, nous extrayons les descripteurs statiques et dynamiques ; nous les donnons au classifieur qui nous renvoie une configuration et nous lançons cette configuration sur i. Notez que le temps passé à extraire les descripteurs et à classifier i est court (moins de 0,1 secondes en moyenne).

5 Évaluation expérimentale.

Approche expérimentale. Nous considérons les 1092 instances décrites en section 3, et l'ensemble d'entraînement est composé des 622 instances difficiles. Nous utilisons un plan de leave-one-out : pour chaque instance i, si i est une instance difficile appartenant à l'ensemble d'entraînement, nous enlevons i de cet ensemble et entraînons le classifieur sur toutes les instances difficiles sauf i ; enfin le classifieur doit classer i.

Comparaison des taux de classification obtenus avec différents ensembles S_k. La configuration apprise d'une instance i est la configuration renvoyée par le classifieur et nous disons que i est *bien classée* si sa configuration apprise est la meilleure configuration pour i parmi les k configurations. La deuxième colonne du tableau 6 donne le pourcentage d'instances bien classées (pour les instances qui ne sont pas résolues en 1 seconde par (CBT,MAC,w)). Nous voyons que le pourcentage décroit quand le nombre de configurations augmente dans S_k, de 82.2% avec deux configurations jusqu'à 51.2% avec 13 configurations. Cependant, le tableau 6 montre également que les configurations apprises des instances mal classées

Rang	1	2	3	4	5	6	7	8	9	10	11	12	13
S_2	82.2	17.8											
S_3	70.7	24.0	5.3										
S_4	68.1	21.8	5.6	4.3									
S_5	57.5	25.7	9.9	5.1	1.6								
S_6	55.4	25.0	9.9	3.8	4.1	1.4							
S_7	54.3	19.6	11.2	6.2	3.6	3.0	1.7						
S_8	53.6	19.9	9.6	5.4	3.2	3.8	2.7	1.4					
S_9	53.5	19.4	8.6	5.3	3.6	2.7	3.3	2.2	0.9				
S_{10}	52.8	18.8	11.0	4.8	2.7	2.7	2.4	2.4	0.9	1.1			
S_{11}	51.7	19.1	9.9	6.2	2.8	2.7	1.7	2.7	1.1	0.8	0.8		
S_{12}	50.9	18.3	10.4	6.2	2.8	1.7	1.6	3.2	0.9	1.6	1.4	0.4	
S_{13}	51.2	17.2	10.1	5.4	4.3	1.4	1.2	2.5	1.9	0.9	1.4	0.8	1.1

Table 6: Taux de classification par rapport au rang des configurations. Pour chaque ensemble S_k et chaque rang $j \in \{1, \ldots, k\}$, le tableau donne le pourcentage d'instances dont la configuration apprise est la j^{me} meilleure parmi les k configurations dans S_k.

Temps	(CBJ,MAC,w)	(CBT,MAC,w)	S_2	S_3	S_4	S_5	S_6	S_7	S_8	S_9	S_{10}	S_{11}	S_{12}	S_{13}
1	38.7	**47.6**	**47.6**	**47.6**	**47.6**	**47.6**	**47.6**	**47.6**	**47.6**	**47.6**	**47.6**	**47.6**	**47.6**	**47.6**
5	52.3	60.6	65.2	67.5	**67.7**	66.7	66.6	66.3	67.3	67.3	67.6	67.2	66.8	66.7
10	57.1	67.2	73.4	**74.5**	74.3	73.7	73.5	72.7	73.7	73.6	74.0	73.7	73.2	73.1
20	64.0	72.6	79.0	**79.9**	79.5	79.2	79.5	78.9	79.2	79.1	78.8	78.8	78.5	78.5
50	73.1	80.0	85.5	**85.7**	85.4	**85.7**	85.3	84.5	85.3	85.0	85.3	85.3	84.6	84.7
100	79.4	84.2	**89.5**	**89.5**	89.1	89.1	88.6	88.0	88.6	88.3	88.8	88.9	88.1	88.2
500	90.2	91.5	94.3	**94.6**	94.2	94.4	94.5	94.0	93.8	94.0	94.5	**94.6**	94.2	94.2
1000	93.1	93.0	**95.8**	95.6	95.2	95.1	95.3	95.0	94.7	95.1	95.3	95.6	95.3	95.2
1800	94.9	94.1	**96.2**	96.1	95.8	95.6	95.8	95.3	95.2	95.4	95.9	96.1	95.6	95.5

Table 7: Pourcentage d'instances résolues selon le temps (en secondes). Pour chaque limite de temps t nous affichons le pourcentage d'instances résolues en moins de t secondes par les deux meilleures configurations ((CBJ,MAC,w) et (CBT,MAC,w)), et par notre sélecteur d'algorithme par instance (avec un ensemble S_k de k configurations). Pour chaque limite de temps, nous soulignons en gras les plus hauts taux de succès parmi toutes les variantes du sélecteur.

correspondent souvent à des configurations qui se comportent bien : étant donné S_k et une instance i, nous ordonnons chaque configuration de S_k de 1 à k selon ses performances sur i (la meilleure étant au rang 1 et la pire au rang k). Par exemple, regardons les résultats pour S_{13} : pour 51.2% des instances, la configuration apprise est la meilleure ; pour 17.2% des instances, c'est la deuxième meilleure ; ...

Comparaison des taux de succès. Le tableau 7 affiche le pourcentage d'instances résolues à différentes limites de temps pour les deux meilleures configurations (CBT,MAC,w) et (CBJ,MAC,w), et pour notre sélecteur d'algorithme par instance avec différents ensembles de configurations de S_2 jusqu'à S_{13}. À 1 seconde, les sélecteurs ont le même taux de succès que (CBT,MAC,w) comme le sélecteur lance d'abord (CBT,MAC,w) pendant 1 seconde. Cependant après 5 secondes, toutes les variantes de notre sélecteur ont de bien meilleures taux de succès que (CBT,MAC,w) et (CBJ,MAC,w). Les meilleures performances sont souvent obtenues par S_3.

La courbe 1 dessine l'évolution du pourcentage d'instances résolues au cours du temps pour les deux meilleures configurations (CBT,MAC,w) et (CBJ,MAC,w) et pour notre sélecteur avec S_3. Elle dessine aussi les résultats d'un solveur virtuel idéal (pour les 13 configurations et pour les 3 configurations de S_3). Cela nous montre que les performances de notre sélecteur sont souvent plus proche des performances du solveur virtuel idéal que des performances de la meilleure configuration.

6 Conclusion

Nous avons étendu le cadre générique de [21] en ajoutant deux mécanismes de backtracking, définissant ainsi un cadre unifié pour comparer 18 configurations correspondant à des approches de l'état de l'art. À notre connaissance, c'est la première fois que des approches basées sur BTD sont comparées avec d'autres stratégies de recherche telles que CBT et DBT, combinées avec FC ou MAC et deux heuristiques de choix de variables (d et w). Les résultats montrent que même si BTD a un taux de succès global bas, il est bien meilleur que d'autres approches sur dif-

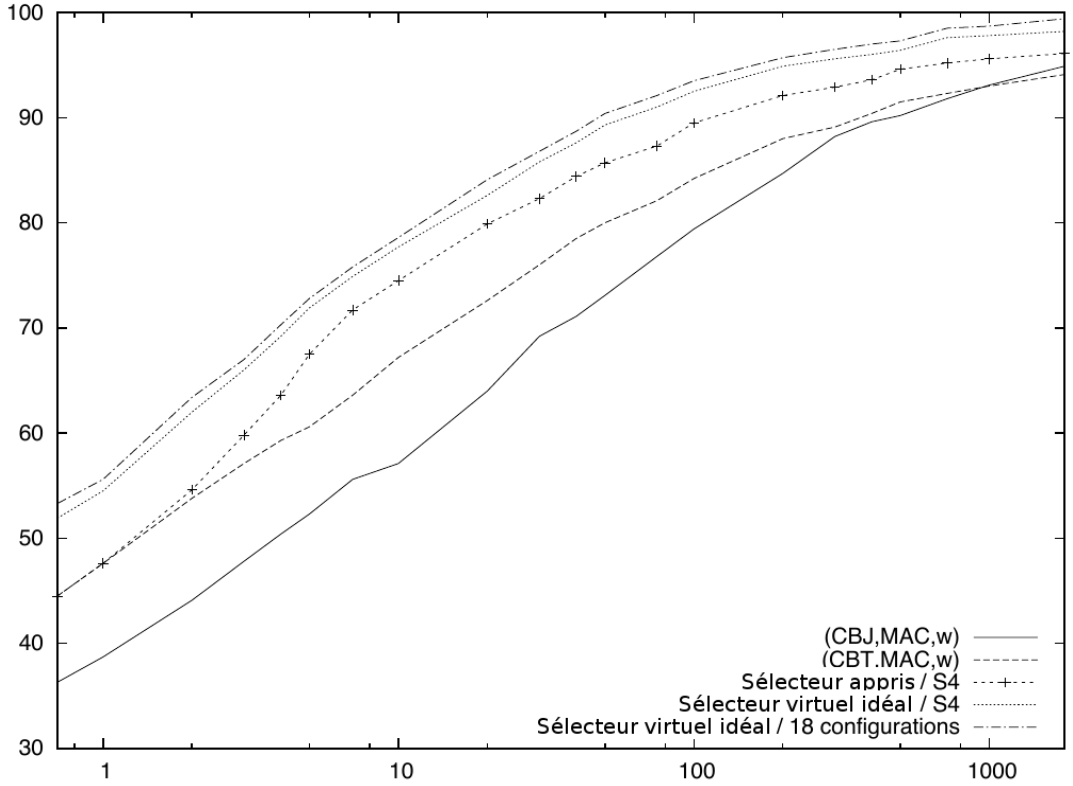


Figure 1: Évolution du pourcentage d'instances résolues au cours du temps (en secondes)

férentes instances. Cependant, la plupart de ces instances sont dans la même classe contenant des instances structurées.

Nous avons décrit un sélecteur d'algorithme par instance qui utilise des techniques d'apprentissage pour choisir une bonne configuration pour résoudre une nouvelle instance. Ce sélecteur est paramétré par le nombre k de configurations pouvant être choisies. Nous avons décrit un algorithme glouton simple pour choisir l'ensemble S_k de k configurations, où le but est de choisir les configurations qui ont des performances complémentaires afin que S_k contienne une bonne configuration pour le plus grand nombre d'instances possible. Le pourcentage d'instances bien classées diminue quand k augmente. Cependant, comme beaucoup d'instances mal classées sont classées dans des configurations plutôt bonnes, le pourcentage d'instances résolues est stable en k.

Dans cette première étude, nous avons extrait des descripteurs pour caractériser les instances et nous voudrions étudier (i) l'utilité des différents descripteurs pour la classification et (ii) la possibilité d'ajouter de nouveaux descripteurs tels que d'autres descripteurs dynamiques récupérés en lançant d'autres algorithmes (comme une recherche gloutonne ou bien de la recherche locale).

Nous allons étendre notre cadre générique en y ajoutant des mécanismes de backtracking comme Decision Repair et des heuristiques de choix de valeurs comme les Impacts.

Nous voulons aussi continuer ce travail pour prédire le temps d'exécution en utilisant des techniques de régression linéaire à l'image de SATzilla. Ce type d'approche prédictive pourrait ensuite être utilisé pour ordonnancer les configurations du portfolio à l'image de CPHydra.

Enfin, notre cadre générique permet de changer dynamiquement de configuration pendant l'étape de résolution. Ainsi nous voulons étendre nos travaux à des configurations dynamiques comme proposé dans [10] ou [22].

References

[1] Roberto Amadini, Maurizio Gabbrielli, and Jacopo Mauro. An empirical evaluation of portfolios approaches for solving csps. In *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, pages 316–324. Springer, 2013.

[2] Fahiem Bacchus. Extending forward checking. In *Principles and Practice of Constraint Programming–CP 2000*, pages 35–51. Springer, 2000.

[3] Andrew B. Baker. The hazards of fancy backtracking. In *AAAI*, pages 288–293, 1994.

[4] Roberto Battiti and Mauro Brunato. *The LION Way: Machine Learning plus Intelligent Optimization.* Lionsolver inc., 2013.

[5] Christian Bessiere and Jean-Charles Régin. Mac and combined heuristics: Two reasons to forsake fc (and cbj?) on hard problems. In *Principles and Practice of Constraint Programming—CP96*, pages 61–75. Springer, 1996.

[6] Christian Bessière and Jean-Charles Régin. Refining the basic constraint propagation algorithm. In *IJCAI*, volume 1, pages 309–315, 2001.

[7] Loïc Blet, Samba Ndojh Ndiaye, and Christine Solnon. Intégration d'une approche structurelle dans un cadre hybride pour la résolution de CSP. In *Actes de la conférence RFIA 2012*, Lyon, France, January 2012. http://hal.archives-ouvertes.fr/hal-00656564.

[8] Frédéric Boussemart, Fred Hemery, Christophe Lecoutre, and Lakhdar Sais. Boosting systematic search by weighting constraints. In *ECAI*, volume 16, page 146, 2004.

[9] Xinguang Chen and Peter Van Beek. Conflict-directed backjumping revisited. *arXiv preprint arXiv:1106.0254*, 2011.

[10] Hani El Sakkout, Mark G Wallace, and E Barry Richards. An instance of adaptive constraint propagation. In *Principles and Practice of Constraint Programming—CP96*, pages 164–178. Springer, 1996.

[11] Eibe Frank, Yong Wang, Stuart Inglis, Geoffrey Holmes, and Ian H Witten. Using model trees for classification. *Machine Learning*, 32(1):63–76, 1998.

[12] Matthew Ginsberg Ginsberg. Dynamic backtracking. *Journal of Artificial Intelligence Research*, 1:25–46, 1993.

[13] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H Witten. The weka data mining software: an update. *ACM SIGKDD Explorations Newsletter*, 11(1):10–18, 2009.

[14] Geoffrey Holmes, Andrew Donkin, and Ian H Witten. Weka: A machine learning workbench. In *Intelligent Information Systems, 1994. Proceedings of the 1994 Second Australian and New Zealand Conference on*, pages 357–361. IEEE, 1994.

[15] P. Jégou, S. N. Ndiaye, and C. Terrioux. Strategies and Heuristics for Exploiting Tree-decompositions of Constraint Networks. In *Inference methods based on graphical structures of knowledge (WIGSK'06), ECAI workshop*, pages 13–18, 2006.

[16] Philippe Jégou, Samba Ndiaye, and Cyril Terrioux. Dynamic heuristics for backtrack search on tree-decomposition of csps. In *IJCAI*, pages 112–117, 2007.

[17] Narendra Jussien, Romuald Debruyne, and Patrice Boizumault. Maintaining arc-consistency within dynamic backtracking. In *Principles and Practice of Constraint Programming–CP 2000*, pages 249–261. Springer, 2000.

[18] Philippe Jégou and Cyril Terrioux. Hybrid backtracking bounded by tree-decomposition of constraint networks. *Artif. Intell.*, 146:43–75, May 2003.

[19] Serdar Kadioglu, Yuri Malitsky, Meinolf Sellmann, and Kevin Tierney. Isac-instance-specific algorithm configuration. In *ECAI*, volume 215, pages 751–756, 2010.

[20] Uffe Kjaerulff. Triangulation of graphs : Algorithms giving small total state space. Technical report, University of Aalborg, 1990.

[21] Christophe Lecoutre, Frederic Boussemart, and Fred Hemery. Backjump-based techniques versus conflict-directed heuristics. In *Tools with Artificial Intelligence, 2004. ICTAI 2004. 16th IEEE International Conference on*, pages 549–557. IEEE, 2004.

[22] Giovanni Di Liberto, Serdar Kadioglu, Kevin Leo, and Yuri Malitsky. Dash: Dynamic approach for switching heuristics. *CoRR*, abs/1307.4689, 2013.

[23] Alan K Mackworth. Consistency in networks of relations. *Artificial intelligence*, 8(1):99–118, 1977.

[24] Yuri Malitsky, Deepak Mehta, and Barry O'Sullivan. Evolving instance specific algorithm configuration. 2013.

[25] Eoin O'Mahony, Emmanuel Hebrard, Alan Holland, Conor Nugent, and Barry O'Sullivan. Using case-based reasoning in an algorithm portfolio for constraint solving. In *Irish Conference on Artificial Intelligence and Cognitive Science*, 2008.

[26] Patrick Prosser. Forward checking with backmarking. In *Constraint Processing, Selected Papers*, pages 185–204, London, UK, 1995. Springer-Verlag.

[27] Francesca Rossi, Peter van Beek, and Toby Walsh. *Handbook of Constraint Programming (Foundations of Artificial Intelligence)*. Elsevier Science Inc., New York, NY, USA, 2006.

[28] Roie Zivan, Uri Shapen, Moshe Zazone, and Amnon Meisels. Retroactive ordering for dynamic backtracking. In *CP*, pages 766–771, 2006.

Une méthode expérimentalement efficace de partition d'une CNF en un MSS et un CoMSS

Éric Grégoire **Jean-Marie Lagniez** **Bertrand Mazure**

CRIL
Université d'Artois & CNRS
rue Jean Souvraz F-62307 Lens, France

{gregoire,lagniez,mazure}@cril.univ-artois.fr

Résumé

Les concepts de MSS (ensemble maximalement satisfiable) et de CoMSS (aussi appelé ensemble minimal de corrections) jouent un rôle important dans différents problèmes et applications d'I.A. Dans ce papier, nous proposons un nouvel algorithme pour scinder une formule CNF en un MSS et son CoMSS correspondant. L'évaluation expérimentale montre que cet algorithme est plus robuste et plus efficace sur la plupart des instances que les techniques existantes.

Abstract

The concepts of MSS (Maximal Satisfiable Subset) and CoMSS (also called Minimal Correction Subset) play a key role in many A.I. approaches and techniques. In this paper, a novel algorithm for partitioning a Boolean CNF formula into one MSS and the corresponding CoMSS is introduced. Extensive empirical evaluation shows that it is more robust and more efficient on most instances than currently available techniques.

1 Introduction

Le concept de MSS (ensemble maximalement satisfiable) joue depuis longtemps un rôle important dans de nombreuses applications de l'I.A. basées sur la logique. En particulier, dans le domaine de la représentation des connaissances, le raisonnement à partir de bases incohérentes est souvent réalisé à l'aide de MSS (par exemple, dans la révision des croyances, la fusion des connaissances, la théorie de l'argumentation ou le raisonnement non monotone ; voir des présentations de synthèse pour chacun de ces domaines dans par exemple [12], [14], [7] et [13]).

L'ensemble complémentaire d'un MSS pour un ensemble incohérent de formules est un CoMSS. Il est souvent appelé ensemble minimal de corrections car il forme un sous-ensemble minimal de formules qui doivent être retirées afin de restaurer la cohérence. Par conséquent, le concept de CoMSS est un paradigme essentiel à la fois au diagnostic à base de modèles (voir les travaux de référence dans [22]) et les techniques de restauration de la cohérence dans les bases de connaissance.

Au-delà du cadre logique, les MSS et CoMSS jouent des rôles similaires dans le domaine des contraintes, lorsque le problème est sur-contraint et qu'il n'admet donc pas de solution (voir par exemple [38, 27] pour les réseaux de contraintes et [9] pour les problèmes d'optimisation liés).

Dans ce papier, nous nous intéressons à ce problème omniprésent dans les différents domaines cités ci-dessus qui consiste donc à scinder une formule booléenne CNF incohérente en un MSS et son CoMSS correspondant, où le critère de maximalité est calculé selon l'inclusion ensembliste (et non la cardinalité). Calculer une telle partition appartient à $\Delta_2^P = P^{NP}$ [36]. Malgré le coût élevé de ce calcul dans le pire cas, plusieurs approches pour extraire une partition (MSS,CoMSS) ont été proposées et se sont montrées viables pour de nombreuses instances.

Clairement, extraire un couple (MSS,CoMSS) est proche du problème MAX-SAT qui consiste à extraire un ensemble de clauses maximalement cohérent pour la cardinalité d'une formule booléenne mise sous forme CNF. Chaque solution de MAX-SAT est un MSS mais tous les MSS ne sont pas nécessairement une solution de MAX-SAT. De manière intéressante, comme suggéré par [30], les algorithmes dédiés au calcul d'un MSS peuvent fournir

une approximation de MAX-SAT lorsque les solutions de ce problème sont hors de portée des algorithmes actuels.

Par ailleurs, les concepts de CoMSS et de MUS (ensemble minimalement incohérent) sont des concepts duaux. Un MUS est un sous-ensemble non satisfiable de clauses tel qu'en supprimer n'importe quel élément rend le sous-ensemble satisfiable. Un MUS peut être calculé à partir d'un « *hitting set* » de l'ensemble des CoMSS d'une formule puisque chaque CoMSS contient au moins une clause de chaque MUS [5, 17, 18, 28].

Dans ce papier, nous proposons un nouvel algorithme pour scinder une formule booléenne CNF insatisfiable en un MSS et un CoMSS. La validation expérimentale que nous avons conduite montre que cette nouvelle approche est particulièrement robuste et plus efficace pour la plupart des instances que les techniques connues pour extraire un MSS ou un CoMSS.

2 Préliminaires logiques et concepts de base

Soit $\mathcal{L}$ le langage de la logique propositionnelle standard construit à partir d'un ensemble fini de variables booléennes et des connecteurs usuels ($\wedge$, $\vee$, $\neg$ et $\rightarrow$ représentant respectivement la conjonction, la disjonction, la négation et l'implication matérielle). Chaque formule de $\mathcal{L}$ peut être représentée par une formule, équivalente du point de vue de la satisfiabilité, sous forme normale conjonctive (CNF). Une CNF est un ensemble de clauses (interprété comme une conjonction) où les clauses sont des disjonctions de littéraux et les littéraux des variables booléennes ou leur négation. Les clauses sont représentées par des lettres minuscules grecques comme α, β, etc. Les ensembles de clauses sont représentés par des lettres grecques majuscules comme Γ, Δ, Σ, etc. Nous notons $\overline{\alpha}$ la négation de la clause α, c'est-à-dire l'ensemble des clauses unitaires (une clause unitaire est constituée d'un seul littéral) données par la négation de chacun des littéraux de α.

Une interprétation $\mathcal{I}$ attribue une valeur dans $\{0, 1\}$ à chaque variable booléenne, et, suivant les règles de compositionnalité standard, à toutes les formules de $\mathcal{L}$. Une formule Γ est cohérente ou satisfiable ssi il existe au moins une interprétation $\mathcal{I}$ qui la satisfait, c'est-à-dire, telle que $\mathcal{I}(\Gamma) = 1$. Pareil $\mathcal{I}$ est alors appelé modèle de Γ et est représenté par l'ensemble des littéraux qu'il satisfait. On dit que α est une déduction logique de Σ, notée $\Sigma \models \alpha$, ssi α est satisfait dans tous les modèles de Σ, ce qui équivaut à ssi $\Sigma \wedge \overline{\alpha}$ n'admet pas de modèle.

SAT est un problème NP-complet qui consiste à vérifier si une formule CNF est satisfiable, c'est-à-dire s'il existe au moins un modèle pour toutes les clauses de la CNF. Dans ce papier, nous faisons souvent référence aux solveurs SAT de type CDCL qui sont actuellement les méthodes complètes les plus efficaces et qui exploitent l'analyse de conflits pour un retour arrière non chronologique et l'apprentissage de clauses (voir par exemple [31, 34, 39, 40, 10, 3, 4, 1, 21]).

Soit Σ une CNF.

Définition 1 (MSS) $\Gamma \subseteq \Sigma$ *est un sous-ensemble maximalement satisfiable (*MSS*) de* Σ *si et seulement si* Γ *est satisfiable et* $\forall \alpha \in \Sigma \setminus \Gamma$, $\Gamma \cup \{\alpha\}$ *n'est pas satisfiable.*

Définition 2 (CoMSS) $\Gamma \subseteq \Sigma$ *est un sous-ensemble minimal de corrections (*MCS *ou* CoMSS*) de* Σ *si et seulement si* $\Sigma \setminus \Gamma$ *est satisfiable et* $\forall \alpha \in \Gamma$, $\Sigma \setminus (\Gamma \setminus \{\alpha\})$ *n'est pas satisfiable. Autrement dit,* Γ *est un* CoMSS *de* Σ *si et seulement si* $\Sigma \setminus \Gamma$ *est un* MSS *de* Σ*.*

Par conséquent, Σ peut toujours être partitionné en un couple (MSS,CoMSS). Évidemment, cette partition n'est pas unique.

Un *core* de Σ est un sous-ensemble de Σ qui n'est pas satisfiable. Les *cores* minimaux pour l'inclusion ensembliste sont appelés des MUS.

Définition 3 (MUS) $\Gamma \subseteq \Sigma$ *est un sous-ensemble minimalement incohérent (*MUS*) de* Σ *si et seulement si* Γ *n'est pas satisfiable et* $\forall \alpha \in \Gamma$, $\Gamma \setminus \{\alpha\}$ *est satisfiable.*

Sous sa forme de base où toutes les clauses sont considérées comme *souples*, c'est-à-dire non obligatoirement satisfaites dans toute solution, MAX-SAT consiste à chercher un sous-ensemble maximalement satisfiable de Σ pour la cardinalité. Comme souligné précédemment, chaque solution de MAX-SAT est un MSS et l'ensemble des CoMSS et des MUS sont des ensembles duaux et peuvent être déduits l'un de l'autre via le calcul de « *hitting sets* ».

Notre algorithme de partition d'une formule CNF est basé sur le concept de *clause de transition*.

Définition 4 (Clause de transition) *Soit* Σ *une* CNF *non satisfiable. Une clause* $\alpha \in \Sigma$ *est une clause de transition de* Σ *si et seulement si* $\Sigma \setminus \{\alpha\}$ *est satisfiable.*

Le concept de clause de transition est fondamental dans de nombreuses approches d'extraction de MUS. Particulièrement, [6] tire avantage de la propriété suivante pour améliorer les performances d'un extracteur de MUS.

Propriété 1 *Soit* Σ *une* CNF *non satisfiable. Si* $\alpha \in \Sigma$ *est une clause de transition alors* α *appartient à tous les* MUS *de* Σ*.*

De manière similaire, dans le cadre de réseaux de contraintes, le concept de contrainte de transition est souvent utilisé pour l'extraction d'un sous-ensemble de contraintes minimalement sur-contraint [23, 15, 16]. Le concept de clause de transition est également un concept clé de notre algorithme de partition.

Algorithm 1: BLS (Basic Linear Search)

input : one CNF Σ
output: one (MSS,CoMSS) partition of Σ

1 $\Pi \leftarrow \Omega \leftarrow \emptyset$;
2 **foreach** $\alpha \in \Sigma$ **do**
3 **if** SAT $(\Pi \cup \alpha)$ **then** $\Pi \leftarrow \Pi \cup \{\alpha\}$;
4 **else** $\Omega \leftarrow \Omega \cup \{\alpha\}$;
5 **return** (Π,Ω) ;

L'algorithme basique pour partitionner une CNF en un couple (MSS,CoMSS) est décrit dans l'Algorithme 1. Il se nomme BLS pour *Basic Linear Search*. Il consiste à ajouter au MSS en construction chaque clause satisfiable avec cet ensemble. De manière évidente, lorsque le nombre de clauses de Σ est égal à n, BLS nécessite n appels à un solveur SAT. Nous présentons cet algorithme uniquement parce que nous pensons qu'il est sans doute le meilleur point de départ pour comprendre la nouvelle approche que nous allons proposer. Un état de l'art des méthodes de partition peut être trouvé par exemple dans [30].

3 CMP : un nouvel algorithme de partition

Algorithm 2: CMP (Computational Method for Partitioning)

input : one CNF Σ
output: one (MSS,CoMSS) partition of Σ

1 $(\Pi, \Psi) \leftarrow$`ApproximatePartition(`Σ`)` ;
2 $\Omega \leftarrow \emptyset$;
3 $\Gamma \leftarrow \Psi$; `// Working subset of` Ψ
4 **while** $\Psi \neq \emptyset$ **do**
5 `extendSatPart(`$\Pi, \Gamma, \Omega, \Psi$`)` ;
6 **if** $\Psi \neq \emptyset$ **then**
7 $\alpha \leftarrow$`selectClause(`Γ`)` ;
8 $(\mathcal{I}, \Delta) \leftarrow$ `solve(`$\Pi \cup (\Gamma \setminus \{\alpha\})$ $\boxed{\cup\, \overline{\alpha}}$`)` ;
  `//Returns a model` $\mathcal{I}$ `if SAT and`
  `//` $(\mathcal{I} = \emptyset$ `and core` $\Delta)$ `otherwise`
9 **if** $\mathcal{I} \neq \emptyset$ **then** `// Transition clause`
10 $\Omega \leftarrow \Omega \cup \{\alpha\}$;
11 $\Pi \leftarrow \Pi \cup \{\beta \in (\Psi \setminus \{\alpha\})$ s.t. $\mathcal{I}(\beta) = 1\}$ $\boxed{\cup\, \overline{\alpha}}$;
12 $\Gamma \leftarrow \Psi \leftarrow \{\beta \in (\Psi \setminus \{\alpha\})$ s.t. $\mathcal{I}(\beta) = 0\}$;
13 **else**
14 $\Gamma \leftarrow \Gamma \setminus \{\alpha\}$;
15 `exploitCore(`$\Pi, \Gamma, \Psi, \Delta, \alpha$`)` ;
16 **return** $(\Pi \cap \Sigma,\Omega)$;

Procedure `ApproximatePartition(`Σ`)`

1 **if** SAT (Σ) **then** `// Satisfiable instance`
2 **return** $(\Sigma, \emptyset)$;
3 **else**
4 Let $\mathcal{P}$ be the progress-saving interpretation delivered by the previous call to the CDCL-SAT solver ;
5 **return** $(\{\beta \in \Sigma$ *s.t.* $\mathcal{P}(\beta)$=1$\}$,$\{\beta \in \Sigma$ *s.t.* $\mathcal{P}(\beta)$=0$\})$;

Le nouvel algorithme de partition de CNF en un MSS et son CoMSS correspondant est appelé CMP pour *Computational Method for Partitioning*. Son ossature est décrite dans l'Algorithme 2, où les instructions encadrées sont optionnelles et seront discutées dans la section 6.

CMP retourne une partition (Π, Ω) de la CNF Σ où Π est un MSS de Σ (et Ω le CoMSS correspondant). À ce niveau, il est important de remarquer les deux points suivants.

1. La boucle de CMP effectue une recherche de clauses de transition qui diffère fondamentalement de celle de BLS. Dans le pire cas, CMP réalise $O(n^2)$ appels à un solveur SAT alors que BLS effectue seulement un nombre linéaire d'appels au solveur. Cependant, notre étude expérimentale montre que même sans les instructions optionnelles encadrées, l'algorithme de base de CMP est plus efficace que tous les autres algorithmes sur de nombreuses instances.
2. La version complète de CMP inclut plusieurs fonctionnalités supplémentaires qui sont encadrées dans l'Algorithme 2. Nous montrons que chacune d'elles ainsi que leur succession permettent d'améliorer les performances de notre approche. Il est à noter que certaines de ces options sont déjà présentes dans d'autres approches comme `extendSatPart`, que d'autres sont exploitées de manière originale (c'est le cas pour le concept que nous appelons « *opposite enforcement on backbones* ») et les dernières constituent de nouveaux concepts (`exploitCore`).

4 CMP : présentation pas à pas

CMP commence par calculer une approximation d'un MSS de Σ via la fonction `ApproximatePartition` (Alg. 2, ligne 1) qui réalise un appel à un solveur de type CDCL pour vérifier la satisfiabilité de Σ. Si Σ est satisfiable, `ApproximatePartition` retourne la partition $(\Sigma, \emptyset)$ et comme $\Psi = \emptyset$, CMP n'entre pas dans la boucle principale et l'algorithme se termine. Dans le cas contraire, `ApproximatePartition` partitionne Σ en un couple (Π, Ψ), où Π est un sous-ensemble du MSS que va calculer l'algorithme. Ce sous-ensemble est obtenu à partir des clauses satisfaites par une interprétation $\mathcal{P}$ retournée par le

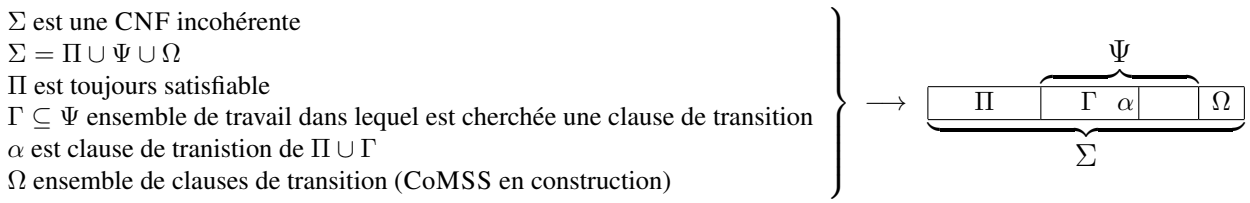


FIGURE 1 – Représentation des différents ensembles de clauses de CMP

solveur CDCL. Cette interprétation correspond à celle du *progress-saving* [37], elle capture la valeur finale des variables et fournit, comme attendu, généralement une bonne approximation d'un MSS de Σ. Π sera modifié au cours de la recherche pour fournir le MSS final tandis que le CoMSS final Ω est d'abord initialisé à l'ensemble vide (Alg. 2, ligne 2). Quand les parties optionnelles de l'algorithme sont activées, Π peut également être enrichi de clauses qui ne sont pas toutes issues de Ψ mais qui peuvent être déduites, ceci afin d'accélérer le temps pris par le processus global. En fait, dans ce cas, l'invariant de boucle est $\Sigma = (\Pi \cap \Sigma) \cup \Psi \cup \Omega$.

Nous laissons de côté pour l'instant les instructions encadrées et étudions la Figure 1. Au début (Alg. 2, ligne 3) et chaque fois qu'une clause est ajoutée à Ω (Alg. 2, ligne 10), le sous-ensemble de travail Γ est initialisé à Ψ (Alg. 2, ligne 12). Tant que chaque clause de Ψ n'a pas été répartie soit dans le CoMSS, soit dans le MSS en construction, l'algorithme cherche une clause $\alpha \in \Psi$ telle que cette clause soit une clause de transition de $\Pi \cup \Gamma \cup \{\alpha\}$ où $\Gamma \cup \{\alpha\} \subseteq \Psi$, c'est-à-dire telle que $\Pi \cup \Gamma \cup \{\alpha\}$ soit non satisfiable alors que $\Pi \cup \Gamma$ l'est. Lorsque qu'une telle clause α est découverte, elle est déplacée dans le CoMSS en construction (Alg. 2, ligne 10). Toutes les clauses de $\Psi \setminus \{\alpha\}$ qui sont satisfaites (respectivement falsifiées) par le modèle trouvé lors du dernier test de satisfiabilité de $\Pi \cup \Gamma$ sont déplacées dans Π (Alg. 2, ligne 11) (respectivement restent dans Ψ (Alg. 2, ligne 12)). Le sous-ensemble de travail Γ est alors initialisé à la nouvelle valeur de l'ensemble Ψ (Alg. 2, ligne 12). Quand α n'est pas une clause de transition (Alg. 2, ligne 14), celle-ci est retirée du sous-ensemble de travail Γ (Alg. 2, ligne 14). La fonction `selectClause` (Alg. 2, ligne 7) choisit une clause α de Γ. Notre implémentation de cette fonction est basée sur l'heuristique bien connue VSIDS [40] et consiste à choisir la clause ayant le plus grand score VSIDS.

5 CMP : preuve d'exactitude

Pour des raisons de place, nous ne donnons ici que les principaux éléments de la preuve accompagnés de quelques commentaires utiles pour aider à comprendre pourquoi CMP délivre les résultats escomptés.

Tout d'abord, il est facile de voir que la boucle se termine toujours. Gardons en mémoire que Γ est initialisé à Ψ (Alg. 2, lignes 3 et 12). Puisque Σ n'est pas satisfiable alors que Π est toujours satisfiable, il existe au moins une clause $\alpha \in \Gamma$ telle que le test de satisfiabilité (Alg. 2, ligne 9) réussisse. Ceci grâce au fait que précédemment pour chaque α dont le test a été infructueux, celui-ci a été retiré de Γ (Alg. 2, ligne 14). Lorsqu'un modèle est trouvé, Ψ est donc réduit (Alg. 2, ligne 12). Le cas extrême est atteint lorsque la satisfiabilité est prouvée alors que Γ est réduit à une clause $\Gamma = \{\alpha\}$.

À présent, expliquons pourquoi l'ensemble final Ω est un CoMSS de Σ. Tout d'abord, considérons la première fois qu'une clause α est déplacée dans Ω (Alg. 2, ligne 10) car elle a été identifiée comme clause de transition de $\Pi \cup \Gamma$. Ceci correspond à la Figure 1, où Ω est égal à l'ensemble vide. La propriété suivante assure que α peut être ajouté à Ω puisque lorsque α est une clause de transition de $\Pi \cup \Gamma$, α appartient à tous les Θ CoMSS de Σ tels que $\Theta \subseteq \Psi \setminus (\Gamma \setminus \{\alpha\})$.

Propriété 2 *Soit Σ une* CNF *non satisfiable telle que $\Sigma = \Pi \cup \Psi$, Π satisfiable, $\Gamma \subseteq \Psi$ et $\alpha \in \Gamma$ une clause de transition de $\Pi \cup \Gamma$. $\forall \Theta \subseteq \Psi \setminus (\Gamma \setminus \{\alpha\})$ tel que $\Theta \in$ CoMSS(Σ), nous avons $\alpha \in \Theta$.*

Preuve. Par l'absurde. Supposons $\exists\, \Theta \in$ CoMSS(Σ) tel que $\Theta \subseteq \Psi \setminus (\Gamma \setminus \{\alpha\})$ et $\alpha \notin \Theta$, c'est-à-dire $\Theta \subseteq \Psi \setminus \Gamma$. Si $\Theta \in$ CoMSS(Σ) alors $\Sigma \setminus \Theta$ est satisfiable. De plus, puisque $\Gamma \subseteq \Psi$ et $\Theta \subseteq \Psi \setminus \Gamma$, nous avons $\Psi = (\Psi \setminus \Gamma) \cup \Gamma = ((\Psi \setminus \Gamma) \setminus \Theta) \cup \Theta \cup \Gamma$. Ainsi, $\Sigma = \Pi \cup \Psi = \Pi \cup \Gamma \cup ((\Psi \setminus \Gamma) \setminus \Theta) \cup \Theta$. Par conséquent, $(\Sigma \setminus \Theta) \supseteq (\Pi \cup \Gamma)$. Comme $\Sigma \setminus \Theta$ est satisfiable, we avons aussi $\Pi \cup \Gamma$ satisfiable ce qui est en contradictoire puisque $\Pi \cup \Gamma$ est non satisfiable (α est une clause de transition de $\Pi \cup \Gamma$). □

Maintenant, grâce à la propriété suivante, il est simple de montrer par induction que pour chaque clause α ajoutée à Ω, α appartient au même CoMSS que les clauses précédemment ajoutées à Ω.

Propriété 3 *Soit Σ une* CNF *non satisfiable. Supposons que $\Sigma = \Pi \cup \Psi$, Π soit satisfiable, $\Gamma \subseteq \Psi$ et $\alpha \in \Gamma$ soit une clause de transition de $\Pi \cup \Gamma$. Si $\Theta \subseteq (\Psi \setminus \Gamma)$ est un* CoMSS *de $\Sigma \setminus \{\alpha\}$ alors $\Theta \cup \{\alpha\}$ est un* CoMSS *de Σ.*

Procedure `extendSatPart` $(\Pi, \Gamma, \Omega, \Psi)$

1 $(\mathcal{I}, \Delta) \leftarrow$ `solve` $(\Pi \cup \{\bigvee \Gamma\})$;
2 **if** $\mathcal{I} \neq \emptyset$ **then**
3 $\quad \Pi \leftarrow \Pi \cup \{\beta \in \Psi$ s.t. $\mathcal{I}(\beta) = 1\}$;
4 $\quad \Gamma \leftarrow \{\beta \in \Psi$ s.t. $\mathcal{I}(\beta) = 0\}$;
5 **else**
6 $\quad \Omega \leftarrow \Omega \cup \Gamma$;
7 $\quad \Gamma \leftarrow \Psi \leftarrow \Psi \setminus \Gamma$;

Preuve. Par l'absurde. Supposons que $(\Theta \cup \{\alpha\})$ n'est pas un CoMSS de Σ. Prouvons que Θ n'est pas non plus un CoMSS de $\Sigma \setminus \{\alpha\}$. $(\Theta \cup \{\alpha\})$ n'est pas un CoMSS de Σ signifie que soit $\Sigma \setminus (\Theta \cup \{\alpha\})$ n'est pas satisfiable (1), soit $(\Theta \cup \{\alpha\})$ n'est pas minimal, c'est-à-dire $(\Theta \cup \{\alpha\})$ est une sur-approximation d'un CoMSS de Σ (2).

(1) $\Sigma \setminus (\Theta \cup \{\alpha\})$ non satisfiable implique que $(\Sigma \setminus \{\alpha\}) \setminus \Theta$ est non satisfiable et que par conséquent Θ n'est pas un CoMSS of $\Sigma \setminus \{\alpha\}$.

(2) $(\Theta \cup \{\alpha\})$ est une sur-approximation d'un CoMSS de Σ signifie qu'il existe un CoMSS Φ de Σ tel que $\Phi \subset (\Theta \cup \{\alpha\})$. Autrement dit, il existe $\beta \in (\Theta \cup \{\alpha\})$, tel que $\beta \notin \Phi$. La Propriété 2 assure que $\alpha \in \Phi$: nous obtenons donc que $\alpha \neq \beta$.

$\Sigma \setminus (\Theta \cup \{\alpha\})$ est satisfiable (puisque $(\Theta \cup \{\alpha\})$ est une sur-approximation d'un CoMSS de Σ). Donc, $\Sigma \setminus (\Theta \cup \{\alpha\} \cup \{\beta\})$ est également satisfiable. Ainsi, $(\Sigma \setminus \{\alpha\}) \setminus (\Theta \setminus \{\beta\})$ est satisfiable et par conséquent Θ n'est pas un CoMSS de $\Sigma \setminus \{\alpha\}$. □

6 CMP : améliorations optionnelles

Nous avons envisagé quatre améliorations à CMP. Elles sont représentées par les instructions encadrées de l'Algorithme 2.

Tout d'abord, tout comme [8] et [30], nous avons exploité l'idée que vérifier la satisfiabilité d'une CNF logiquement affaiblie où certaines clauses sont remplacées par leur disjonction (notée $\bigvee \Gamma$) pouvait s'avérer informatif. Dans le cas où $\Pi \cup \{\bigvee \Gamma\}$ est non satisfiable, $\Pi \cup \Gamma$ est aussi non satisfiable. Dans le cas où il existe un modèle, l'ensemble des clauses de Γ qui sont satisfaites par le modèle trouvé de $\Pi \cup \{\bigvee \Gamma\}$ peuvent être déplacées dans Π. C'est le rôle de la fonction `extendSatPart` à la ligne 5 de l'Algorithme 2. Cette fonctionnalité joue un rôle important dans l'efficacité de la méthode CLD proposée dans [30].

La seconde amélioration (Alg. 2, ligne 11) concerne les littéraux du « *backbone* » (voir [33] et plus récemment [25]) ; cela a également été exploité dans les procédures d'extraction de CoMSS de [30]. $\overline{\alpha}$ est l'ensemble de clauses unitaires composées des littéraux opposés à ceux

Procedure `exploitCore` $(\Pi, \Gamma, \Psi, \Delta, \alpha)$

1 **if** $\Delta \cap \overline{\alpha} = \emptyset$ **then** $\Gamma \leftarrow \Gamma \cap \Delta$;
2 **elsif** $\Delta \cap \Gamma = \emptyset$ **then**
3 $\quad \Pi \leftarrow \Pi \cup \{\alpha\}$;
4 $\quad \Psi \leftarrow \Psi \setminus \{\alpha\}$;
5 **else** $\Pi \leftarrow \Pi \cup \{\Delta \setminus \overline{\alpha} \vDash \alpha\}$;

de α. Lorsque α est prouvé appartenir au CoMSS que l'on calcule, cela implique que $\overline{\alpha}$ est une conséquence logique de l'ensemble courant Π (et, par monotonie, de tous ses sur-ensembles). Ces clauses unitaires apprises sont ajoutées dans Π (Alg. 2, ligne 11) dans l'espoir d'accélérer les futurs tests de satisfiabilité des sur-ensembles de Π.

Une autre exploitation originale des ces clauses unitaires peut être réalisée lors du test de satisfiabilité pour savoir si α est une clause de transition (Alg. 2, ligne 8). Lorsqu'un modèle est trouvé, $\Pi \cup \Gamma \setminus \{\alpha\}$ est satisfiable alors que $\Pi \cup \Gamma$ ne l'est pas. Un résultat identique peut être obtenu en testant la satisfiabilité de $\Pi \cup (\Gamma \setminus \{\alpha\}) \cup \{\overline{\alpha}\}$, où l'ensemble des clauses unitaires $\overline{\alpha}$ est ajouté pour accélérer le test. À notre connaissance, cette utilisation des littéraux du backbone dans un ensemble de prémisses plus fort qui n'est pas uniquement cohérent avec $\overline{\alpha}$ mais qui implique $\overline{\alpha}$, est nouvelle dans la recherche de CoMSS.

Nous appelons cette option « *opposite enforced* », en abrégé *oe*. De manière intéressante, CMP autorise ce renforcement car la clause de transition est trouvée dans le cas où le test de satisfiabilité devient positif alors que les approches traditionnelles identifient une clause de transition lorsque l'incohérence est atteinte.

La quatrième amélioration correspond à la fonction `exploitCore` (Alg. 2, ligne 15) qui affine de manière originale Γ étant donné Δ le core calculé lorsque le test de satisfiabilité a échoué. Trois cas peuvent être distingués.

1. Quand Δ ne contient aucune clause de $\overline{\alpha}$, Γ peut être affecté à l'intersection ensembliste avec Δ puisque l'on pourra toujours trouver une clause de transition dans cette intersection.
2. Dans le cas contraire, si aucune clause de Γ n'appartient à Δ, alors cela implique que Δ est constitué uniquement de clauses de $\overline{\alpha}$ et Π. Par conséquent, $\Pi \wedge \overline{\alpha}$ est insatisfiable, c'est-à-dire, α est une conséquence logique de Π. Ainsi, α peut être déplacé dans la partie satisfiable Π.
3. Sinon, Δ est construit à partir de clauses de $\overline{\alpha}$ et Γ (et peut-être également de clauses de Π). Dans ce cas, comme au moins une clause de $\overline{\alpha}$ est dans Δ et que Δ est non satisfiable, nous obtenons que $(\Delta \setminus \overline{\alpha}) \wedge \overline{\alpha}$ est non satisfiable, c'est-à-dire que α est une conséquence logique de $\Delta \setminus \overline{\alpha}$. Bien que, cela ne puisse être représenté sous la forme d'un ensemble de clauses de manière directe, cette information peut être ex-

ployée et implémentée facilement (sans coût significatif en espace) en utilisant des sélecteurs de clauses comme cela est fait dans [35]. Chaque clause β de Σ est enrichie d'un nouveau littéral qui lui est propre, que l'on appelle sélecteur et que l'on note λ_β. β est remplacée par $\beta \vee \neg\lambda_\beta$ dans Σ. Une clause est active (c'est-à-dire appartient à) dans un ensemble de clauses si son sélecteur est affecté à 1. Avec ces sélecteurs, $\Delta \setminus \overline{\alpha} \models \alpha$ peut être représenté par la clause $(\neg\lambda_{\delta_1} \vee \cdots \vee \neg\lambda_{\delta_m} \vee \lambda_\alpha)$ où δ_i ($i \in [1..m]$) sont les m clauses de $\Delta \setminus \overline{\alpha}$. Lorsque toutes les clauses de $\Delta \setminus \overline{\alpha}$ sont actives, alors α sera également activé.

Il est simple de montrer que toutes ces améliorations ne remettent pas en question l'exactitude de CMP.

7 Travaux connexes

La gestion de systèmes sur-contraints à travers la recherche d'un ensemble maximalement satisfiable ou d'un ensemble minimal de corrections est un sujet de recherche extrêmement actif en I.A. (voir par exemple les travaux pionniers de [32]). Dans le cadre général des réseaux de contraintes, une approche de référence appelée QUICKXPLAIN est décrite dans [24]. Dans le même cadre, des techniques plus performantes peuvent être trouvées dans [23, 20, 19].

Dans le but de résoudre un réseau de contraintes ou d'en extraire ses MSS ou CoMSS, il est souvent plus efficace d'encoder le réseau sous la forme d'un ensemble de clauses booléennes et de bénéficier des progrès réalisés pour les solveurs SAT (à condition que la taille de la formule booléenne n'explose pas). Dans le domaine SAT, beaucoup d'approches ont été proposées pour calculer des MSS et des CoMSS. Les approches les plus anciennes reposent sur des solveurs SAT basés sur DPLL (voir par exemple [8]) et sont relativement inefficaces comparativement aux approches plus récentes [28] basées sur MAX-SAT. Comme souligné dans [30], les dernières approches sont aussi plus performantes que les différents algorithmes basés sur des appels itératifs à des solveurs SAT comme [5]. Beaucoup de recherches ont également été conduites dans le domaine du diagnostic à base de modèles. Notamment, une approche récente, appelée `FastDiag`, abrégé BFD pour « *Basic FastDiag* » [11], a adpaté QUICKXPLAIN de [24] au cadre booléen et se montre particulièrement compétitif.

Récemment [30] a comparé expérimentalement les meilleures approches d'extraction de MSS et de CoMSS dans le cadre booléen, à savoir les algorithmes BFD, BLS (décrit dans l'Algorithme 1), EFD et ELS pour « *Enhanced FastDiag* » et « *Enhanced Linear Search* » qui intègrent les améliorations discutées dans [30]. Les mêmes auteurs ont aussi proposé un nouvel algorithme pour calculer des CoMSS qui s'est montré expérimentalement plus efficace et plus robuste que toutes les autres méthodes mentionnées ci-dessus. Principalement, cet algorithme, appelé CLD, extrait un CoMSS en utilisant itérativement le principe présenté dans `extendSatPart` (et aussi prenant en compte les « *backbones* », parmi d'autres choses). Les auteurs montrent que CLD surpasse expérimentalement toutes les autres méthodes.

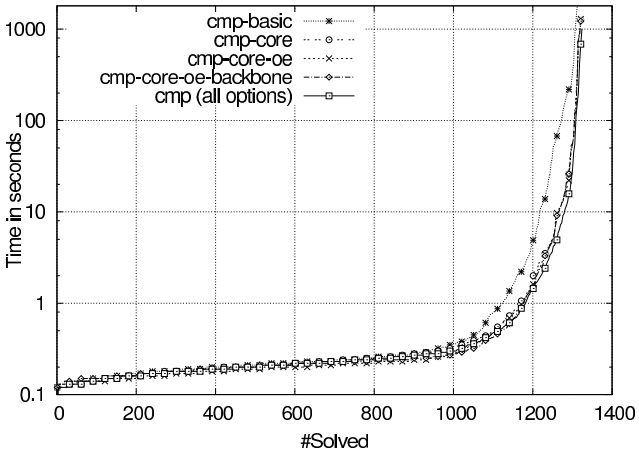


FIGURE 2 – Comportement des différentes variantes de CMP sur les instances SAT et MAX-SAT

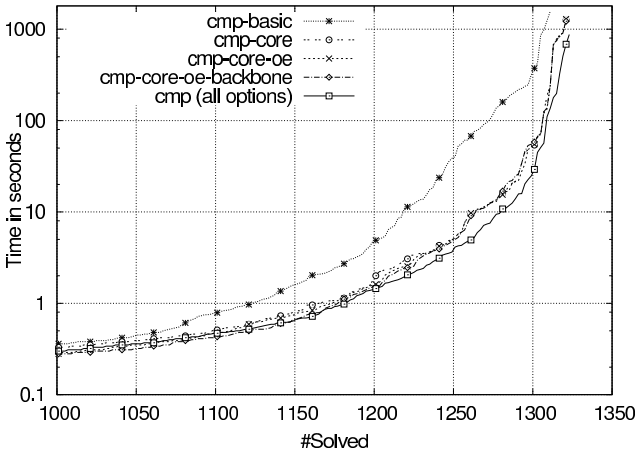


FIGURE 3 – Zoom sur la Figure 2

Bien qu'il implante les améliorations provenant des « *backbones* » et de `extendSatPart`, CMP est de nature très différente de par sa recherche d'une clause de transition plutôt que par une itération de `extendSatPart`. De plus, CMP inclut de nouvelles améliorations comme « *opposite enforced* » et `exploitCore`.

8 Résultats expérimentaux

Toutes les expérimentations ont été conduites sur des processeurs Intel Xeon E5-2643 (3.30GHz) avec 7.6Go de mémoire sous un système Linux CentOS. Le temps maxi-

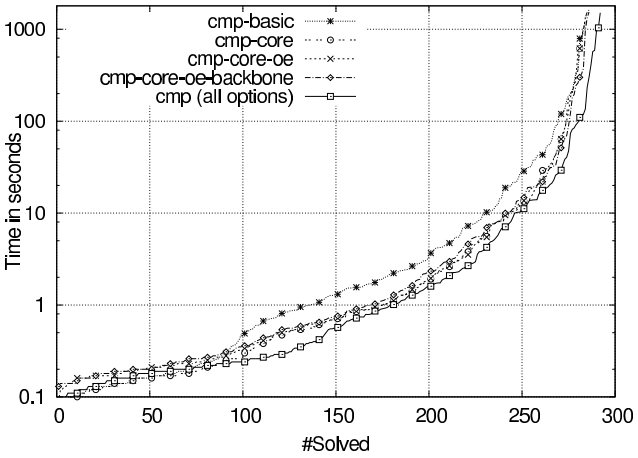


FIGURE 4 – Comportement des différentes variantes de CMP sur les instances MUS

	SAT/MAX-SAT inst.	MUS inst.
Nombre d'instances	1343	295
CMP-*basic*	1312	285
CMP-*core*	1321	285
CMP-*core-oe*	1321	286
CMP-*core-oe-backbone*	1322	286
CMP (all options)	1327	292

TABLE 1 – Nombre d'instances partitionnées

mal de chaque calcul a été limité à 30 minutes.

Deux séries de benchmarks ont été utilisées. La première est constituée des 1343 instances utilisées et référencées dans [30]. Elle comprend des instances industrielles de petite taille issues des compétitions SAT `www.satcompetition.org` et d'instances structurées issues des évaluations MAX-SAT`maxsat.ia.udl.cat:81`. Nous avons enrichi cet ensemble de benchmarks par un second ensemble formé de toutes les instances utilisées lors de la compétition MUS organisée en parallèle de la compétition SAT. Toutes les courbes présentées donnent le temps CPU nécessaire pour partitionner un nombre donné d'instances pour chacune des approches. Les partitions calculées ne sont pas toutes les mêmes et peuvent différer d'une approche à l'autre.

CMP a été programmé en C++. MINISAT [10] a été choisi comme solveur SAT. CMP et toutes les données nécessaires à la reproduction de ces expérimentations sont disponibles à l'adresse `www.cril.univ-artois.fr/documents/cmp/`.

Tout d'abord, les gains des quatre parties optionnelles de CMP ont été évalués.

La version basique de CMP, c'est-à-dire l'Algorithme 2 sans aucune des instructions encadrées a été renommé CMP-*basic*. CMP-*basic* a été ensuite enrichi des différentes options de manière incrémentale. L'ajout seul de `exploitCore` (Alg. 2, ligne 15) donne naissance à CMP-*core* ; l'ajout de « *opposite enforced* » (Alg. 2, ligne 8) conduit à la version appelée CMP-*core-oe*. CMP-*core-oe-backbone* est obtenu en activant les littéraux « *backbones* » (Alg. 2, ligne 11) . À partir de maintenant, CMP dénote l'Algorithme 2 avec toutes les options, ce qui inclut donc également `extendSatPart` (Alg. 2, ligne 5).

Les Figures 2 et 4 montrent les améliorations graduelles de chacune des options lorsque ces options sont prises en compte de manière cumulative : chaque nouvelle option permet d'obtenir un gain en efficacité. La Figure 3 est un zoom sur la partie droite des courbes de la Figure 2. La Table 1 montre que le nombre d'instances qui ont pu être partitionnées augmente également en fonction du nombre d'options activées. Visiblement, l'option `exploitCore` fournit les plus importants gains à la fois en temps d'exécution mais également en nombre d'instances partitionnées. Il est à noter que l'incrémentalité des apports de chacune des options a été observée quel que soit l'ordre de combinaison choisi.

Nous avons ensuite comparé CMP avec les approches existantes et mentionnées précédemment, à savoir les algorithmes : BFD, BLS, CLD, EFD et ELS qui sont décrits dans [30] et implémentés dans l'outil MCS_{LS} `logos.ucd.ie/wiki/doku.php?id=mcsls`. La version 2 de CAMUS (`sun.iwu.edu/~mliffito/camus/` [28, 29]), que nous nommons CAMUS 2, a également été testée, mais uniquement sur les benchmarks SAT et MAX-SAT : ce programme qui cherche à calculer tous les MUS n'est capable que de partitionner qu'un petit nombre d'instances. La Table 2 montre que la version CMP-*basic* est capable de paritionner plus d'instances que toutes les autres méthodes. Les Figures 5 and 6 donnent une représentation visuelle des performances de chacune des méthodes. Nous avons également tracé la courbe du « *Virtual Best Solver* » (VBS), qui représente pour chaque instance la méthode la plus rapide pour trouver une partition. De manière incontestable, CMP obtient les meilleurs performances et se trouve très proche du VBS. Pour chaque méthode, la Table 2 donne le nombre d'instances pour lesquelles une partition a pu être trouvée. Le meilleur score est également détenu par CMP.

9 Discussion and perspectives

Les expérimentations intensives que nous avons conduites montrent que CMP est plus robuste que les approches précédemment proposées et permet de calculer une partition pour un plus grand nombre d'instances booléennes. Évidemment, cela n'implique pas que CMP soit le plus efficace pour toutes les instances. À cet égard, il convient de rappeler que la complexité dans le pire cas de CMP est plus importante que celle de l'approche BLS par exemple. En effet, lorsque n est le nombre de clauses

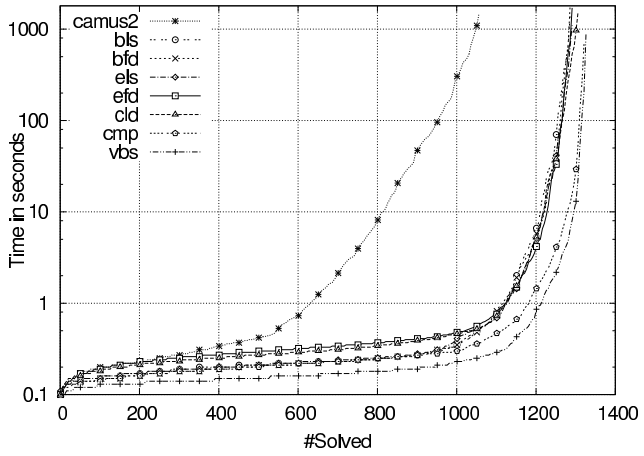


FIGURE 5 – Comparaison sur les instances SAT et MAX-SAT

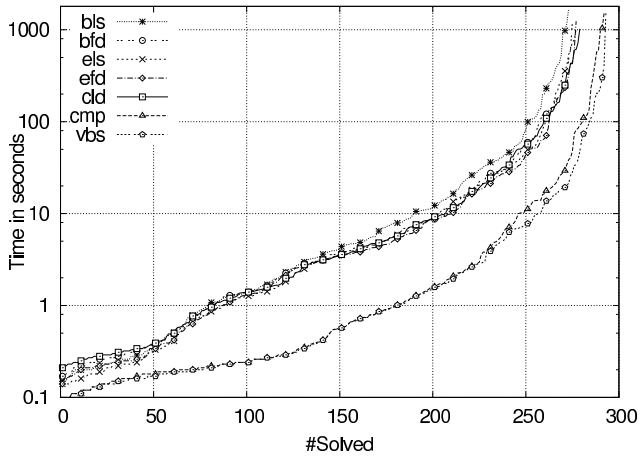


FIGURE 6 – Comparaison sur les instances MUS

de l'instance, CMP nécessite $O(n^2)$ appels à un solveur SAT dans le pire cas alors que BLS n'en nécessite qu'un nombre linéaire (en l'occurrence n). Néanmoins, nous avons constaté que le nombre d'appels au solveur SAT faits par CMP reste toujours très significativement plus petit que n pour toutes les instances pour lesquelles il a réussi à calculer une partition (voir Figure 7).

Nous envisageons plusieurs poursuites de ce travail. Premièrement, la méthode pourrait gagner en efficacité en utilisant les outils d' « *incremental* SAT » [26, 2]. Deuxièmement, CMP peut être utilisé pour approximer MAX-SAT lorsque le calcul exact est hors de portée des algorithmes actuels. À cet égard, bien que toutes les options de CMP ne ie pas directement exportables dans les algorithmes dédiés à MAX-SAT, nous pensons que la manière de calculer des MSS dans CMP peut conduire à de nouvelles pistes pour le calcul de MAX-SAT. CMP est capable de fournir des solutions approximatives pour MAX-SAT, il serait intéressant de voir comment il est possible de faire de même

	SAT/MAX-SAT inst.	MUS inst.
Nombre d'instances	1343	295
BLS	1287	273
BFD	1287	276
ELS	1293	277
EFD	1291	277
CLD	1307	279
CMP-*basic*	1312	285
CMP (all options)	1327	292
VBS	1327	293

TABLE 2 – Nombre d'instances partitionnées par méthode

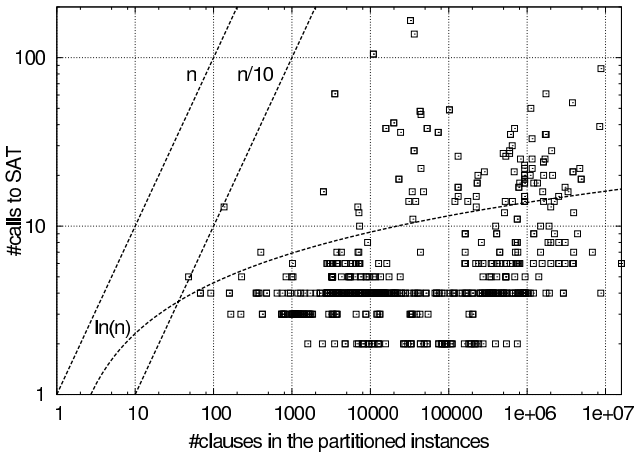


FIGURE 7 – Nombre d'appels à un solveur SAT effectués par CMP

pour *weighted* our *partial* MAX-SAT. Enfin, il serait intéressant d'étendre CMP au problème de l'énumération des CoMSS (ou MSS) et d'étudier s'il est possible d'améliorer les résultats pratiques récemment obtenus pour ce problème [30].

Références

[1] Gilles Audemard, Jean-Marie Lagniez, Bertrand Mazure, and Lakhdar Saïs. On freezing and reactivating learnt clauses. In *Proceedings of the 14th International Conference on Theory and Application of Satisfiability Testing (SAT'11)*, pages 188–200. Springer, 2011.

[2] Gilles Audemard, Jean-Marie Lagniez, and Laurent Simon. Improving Glucose for incremental SAT solving with assumptions : Application to MUS extraction. In *Proceedings of the 16th International Conference on Theory and Applications of Satisfiability Testing (SAT'13)*, volume 7962 of *Lecture Notes in Computer Science*, pages 309–317. Springer, 2013.

[3] Gilles Audemard and Laurent Simon. Predicting learnt clauses quality in modern sat solvers. In *Proceedings of the 21st International Joint Conference on Artifical Intelligence*, IJCAI'09, pages 399–404, 2009.

[4] Gilles Audemard and Laurent Simon. Refining restarts strategies for sat and unsat. In *Principles and Practice of Constraint Programming - 18th International Conference (CP'12)*, volume 7514 of *Lecture Notes in Computer Science*, pages 118–126. Springer, 2012.

[5] James Bailey and Peter J. Stuckey. Discovery of minimal unsatisfiable subsets of constraints using hitting set dualization. In *Proceedings of the 7th International Symposium on Practical Aspects of Declarative Languages (PADL'2005)*, volume 3350 of *Lecture Notes in Computer Science*, pages 174–186. Springer, 2005.

[6] Anton Belov and João Marques-Silva. Accelerating MUS extraction with recursive model rotation. In *Proceedings of the International Conference on Formal Methods in Computer-Aided Design (FMCAD'11)*, pages 37–40. FMCAD Inc., 2011.

[7] Philippe Besnard and Anthony Hunter. *Elements of Argumentation*. The MIT Press, 2008.

[8] Elazar Birnbaum and Eliezer L. Lozinskii. Consistent subsets of inconsistent systems : structure and behaviour. *J. Exp. Theor. Artif. Intell.*, 15(1) :25–46, 2003.

[9] John W. Chinneck. *Feasibility and Infeasibility in Optimization : Algorithms and Computational Methods*, volume 118 of *International Series in Operations Research & Management Science*. Springer, 2008.

[10] Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In *Proceedings of the 6th International Conference on Theory and Applications of Satisfiability Testing (SAT'03). Selected Revised Papers*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2004.

[11] Alexander Felfernig, Monika Schubert, and Christoph Zehentner. An efficient diagnosis algorithm for inconsistent constraint sets. *AI EDAM*, 26(1) :53–62, 2012.

[12] Eduardo L. Fermé and Sven Ove Hansson. AGM 25 years - twenty-five years of research in belief change. *J. Philosophical Logic*, 40(2) :295–331, 2011.

[13] M.L. Ginsberg. *Readings in nonmonotonic reasoning*. M. Kaufmann Publishers, 1987.

[14] Éric Grégoire and Sébastien Konieczny. Logic-based approaches to information fusion. *Information Fusion*, 7(1) :4–18, 2006.

[15] Éric Grégoire, Jean-Marie Lagniez, and Bertrand Mazure. Improving MUC extraction thanks to local search. *CoRR*, abs/1307.3585, 2013.

[16] Éric Grégoire, Jean-Marie Lagniez, and Bertrand Mazure. Questioning the importance of WCORE-like minimization steps in MUC-finding algorithms. In *Proceedings of the 25th IEEE International Conference on Tools with Artificial Intelligence (ICTAI'13)*, pages 923–930, 2013.

[17] Éric Grégoire, Bertrand Mazure, and Cédric Piette. Boosting a complete technique to find MSS and MUS thanks to a local search oracle. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI'07)*, pages 2300–2305, 2007.

[18] Éric Grégoire, Bertrand Mazure, and Cédric Piette. Local-search extraction of MUSes. *Constraints*, 12(3) :325–344, sep 2007.

[19] Éric Grégoire, Jean-Marie Lagniez, and Bertrand Mazure. Questioning the importance of wcore-like minimization steps in muc-finding algorithms. In *25th International Conference on Tools with Artificial Intelligence(ICTAI'13)*, pages 923–930. IEEE Press, 2013.

[20] Éric Grégoire, Bertrand Mazure, and Cédric Piette. On finding minimally unsatisfiable cores of csps. *International Journal on Artificial Intelligence Tools (IJAIT)*, 17(4) :745 – 763, aug 2008.

[21] Long Guo and Jean-Marie Lagniez. Dynamic polarity adjustment in a parallel SAT solver. In *Proceedings of the 23rd IEEE International Conference on Tools with Artificial Intelligence (ICTAI'11)*, pages 67–73, 2011.

[22] Walter Hamscher, Luca Console, and Johan de Kleer, editors. *Readings in Model-Based Diagnosis*. Morgan Kaufmann, 1992.

[23] Fred Hemery, Christophe Lecoutre, Lakhdar Saïs, and Frédéric Boussemart. Extracting MUCs from constraint networks. In *Proc. of the 17th European Conference on Artificial Intelligence (ECAI'06)*, volume 141 of *Frontiers in Artificial Intelligence and Applications*, pages 113–117. IOS Press, 2006.

[24] Ulrich Junker. QUICKXPLAIN : Preferred explanations and relaxations for over-constrained problems. In *Proceedings of the 9th National Conference on Artificial Intelligence (AAAI'04)*, pages 167–172. AAAI Press, 2004.

[25] Philip Kilby, John K. Slaney, Sylvie Thiébaux, and Toby Walsh. Backbones and backdoors in satisfiability. In *Proceedings, The 20th National Conference on Artificial Intelligence (AAAI'05)*, pages 1368–1373. AAAI Press / The MIT Press, 2005.

[26] Jean-Marie Lagniez and Armin Biere. Factoring out assumptions to speed up mus extraction. In *Proceedings of the 16th International Conference on Theory and Applications of Satisfiability Testing (SAT'13)*, volume 7962 of *Lecture Notes in Computer Science*, pages 276–292. Springer, 2013.

[27] Christophe Lecoutre. *Constraint Networks : Techniques and Algorithms*. Wiley, 2009.

[28] Mark H. Liffiton and Karem A. Sakallah. Algorithms for computing minimal unsatisfiable subsets of constraints. *J. Autom. Reasoning*, 40(1) :1–33, 2008.

[29] Mark H. Liffiton and Karem A. Sakallah. Generalizing core-guided Max-SAT. In *Proceedings of the 12th International Conference on Theory and Applications of Satisfiability Testing (SAT'09)*, volume 5584 of *Lecture Notes in Computer Science*, pages 481–494. Springer, 2009.

[30] Joao Marques-Silva, Federico Heras, Mikolás Janota, Alessandro Previti, and Anton Belov. On computing minimal correction subsets. In *Proceedings of the 23rd International Joint Conference on Artificial Intelligence (IJCAI'13)*, 2013.

[31] Joao Marques-Silva and Karem A. Sakallah. GRASP : a new search algorithm for satisfiability. In *Proceedings of the 1996 IEEE/ACM International Conference on Computer-aided Design (ICCAD'96)*, pages 220–227. IEEE Computer Society, 1996.

[32] Pedro Meseguer, Noureddine Bouhmala, Taoufik Bouzoubaa, Morten Irgens, and Martí Sánchez. Current approaches for solving over-constrained problems. *Constraints*, 8(1) :9–39, 2003.

[33] Rémi Monasson, Riccardo Zecchina, Scott Kirkpatrick, Bart Selman, and Lidror Troyansky. Determining computational complexity from characteristic 'phase transitions'. *Nature*, 400 :133, 1999.

[34] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff : Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference (DAC'01)*, pages 530–535. ACM, 2001.

[35] Yoonna Oh, Maher N. Mneimneh, Zaher S. Andraus, Karem A. Sakallah, and Igor L. Markov. AMUSE : A minimally-unsatisfiable subformula extractor. In *Proceedings of the 41th Design Automation Conference (DAC'04)*, pages 518–523. ACM, 2004.

[36] Christos H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1993.

[37] K. Pipatsrisawat and A. Darwiche. A lightweight component caching scheme for satisfiability solvers. In *Proceedings of the 10th International Conference on Theory and Applications of Satisfiability Testing (SAT'07)*, volume 4501 of *Lecture Notes in Computer Science*, pages 294–299. Springer, 2007.

[38] F. Rossi, P. van Beek, and T. Walsh, editors. *Handbook of Constraint Programming*. Elsevier, 2006.

[39] L. Zhang, C. Madigan, M. Moskewicz, and S. Malik. Efficient conflict driven learning in boolean satisfiability solver. In *Proc. of the Proceedings of IEEE/ACM International Conference on Computer Design (ICCAD'01)*, pages 279–285, 2001.

[40] Lintao Zhang and Sharad Malik. The quest for efficient boolean satisfiability solvers. In *Proceedings of the 14th International Conference on Computer Aided Verification (CAV'02)*, volume 2404 of *Lecture Notes in Computer Science*, pages 17–36. Springer, 2002.

Microstructures pour CSP d'arité quelconque * †

Achref El Mouelhi **Philippe Jégou** **Cyril Terrioux**

Aix-Marseille Université, LSIS UMR 7296
Avenue Escadrille Normandie-Niemen
13397 Marseille Cedex 20 (France)
{achref.elmouelhi, philippe.jegou, cyril.terrioux}@lsis.org

Résumé

Il est admis depuis longtemps que la notion de microstructure de CSP offre un cadre graphique utile pour l'étude de propriétés théoriques. Malheureusement, cette représentation graphique est restreinte aux seuls CSP binaires. Aussi, dans cette contribution, nous proposons d'étendre cette notion fondée sur les graphes, aux CSP d'arité quelconque. Cette approche qui évite le recours aux hypergraphes permet ainsi de disposer d'une littérature bien plus riche comme l'est celle de la Théorie (Algorithmique) des Graphes en comparaison à celle offerte par la Théorie des Hypergraphes. Nous introduisons trois définitions possibles de microstructures basées sur les codages binaires d'un CSP non binaire. Nous montrons également que ces représentations peuvent former un nouvel outil théorique pour généraliser certains résultats introduits au niveau des classes polynomiales de CSP binaires. Nous pensons que ces représentations pourraient être utiles à la communauté pour des développements de nature théorique, d'une part pour l'extension de résultats existants, mais aussi pour produire des résultats originaux pour les CSP d'arité quelconque.

1 Préliminaires

Les problèmes de satisfaction des contraintes (CSP pour Constraint Satisfaction Problems en anglais [29]) présente un moyen efficace pour la modélisation et la résolution de certains problèmes en Recherche Opérationnelle et en Intelligence Artificielle.

Formellement, une instance CSP est un triplet $P = (X, D, C)$, où $X = \{x_1, ..., x_n\}$ est un ensemble fini de n variables, $D = \{d_1, ..., d_n\}$ est un ensemble de domaines finis de valeurs, un pour chaque variable et $C = \{c_1, ..., c_e\}$ est un ensemble fini de contraintes. Chaque contrainte c_i est un couple $(S(c_i), R(c_i))$, où $S(c_i) = \{x_{i_1}, ..., x_{i_k}\} \subseteq X$ est la portée (aussi appelée « scope ») de la contrainte. $|S(c_i)|$ est l'arité de la contrainte c_i, c'est-à-dire, le nombre de variables sur lesquelles la contrainte c_i porte. $R(c_i) \subseteq d_{i_1} \times ... \times d_{i_k}$ est sa relation de compatibilité, chaque combinaison de valeurs d'une relation c_i est appelée tuple et sera notée t.

Nous supposons que toute variable apparaît dans la portée d'au moins une contrainte et que toutes les relations sont en extension, c'est-à-dire que les relations contiennent la liste des tuples autorisés. Si la contrainte est d'arité deux, alors elle est dite binaire et elle sera notée c_{ij} avec $S(c_{ij}) = \{x_i, x_j\}$. Si toutes les contraintes sont binaires, le CSP est dit *binaire*. La structure d'un réseau de contraintes est représentée par un hypergraphe (un graphe pour le cas binaire) noté $H(P) = (X, C)$. Les sommets de $H(P)$ correspondent aux variables et les hyperarêtes aux portées des contraintes. Une affectation d'un sous-ensemble de X est dite cohérente (ou consistante) si elle ne viole aucune contrainte. Une solution est une affectation cohérente portant sur toutes les variables de X. Le problème du test d'existence d'une solution est NP-complet. De ce fait, de nombreux travaux ont été réalisés pour rendre la résolution des instances plus efficace en pratique, en utilisant des versions de backtrack améliorées, des techniques de filtrage par consistance partielle, et des heuristiques d'ordonnancement des variables, des valeurs, voire des contraintes. Une seconde piste intéressante consiste en l'étude des *classes polynomiales* (dites aussi *traitables*, de l'anglais « tractable »). Une classe polynomiale est un ensemble infini d'instances défini par des restrictions soit

*Ce travail est soutenu par l'Agence Nationale de la Recherche dans le cadre du projet TUPLES (ANR-2010-BLAN-0210).

†Ce travail a été présenté dans sa version anglaise lors de la conférence SARA 2013 [14]

sur la portée des contraintes ou du graphe de contraintes, soit sur les relations associées aux contraintes. Les instances d'une telle classe peuvent être résolues en temps polynomial, et souvent la communauté s'accorde pour imposer que le test d'appartenance à une telle classe soit réalisable également en temps polynomial. Par exemple, les CSP binaires ayant un graphe de contraintes acyclique sont résolubles en temps linéaire [16]. Ce résultat a été par la suite généralisé aux CSP d'arité quelconque [20]. Des méthodes de résolution (dont notamment Tree Clustering [11]) ont été fondées sur de telles propriétés. Certaines ont montré l'intérêt pratique de ce type d'approches [25].

La Théorie des Graphes et la Théorie des Hypergraphes ont aussi permis de mettre en évidence des classes polynomiales sur la base de restrictions sur les relations de compatibilité pour le cas des CSP binaires. Elles se sont parfois appuyées sur une représentation appelée *graphe de microstructure* :

Définition 1 (microstructure) *Étant donné un CSP* $P = (X, D, C)$*, la* microstructure *de* P *est un graphe non orienté* $\mu(P) = (V, E)$ *avec :*
- $V = \{(x_i, v_i) : x_i \in X, v_i \in d_i\}$,
- $E = \{ \{(x_i, v_i), (x_j, v_j)\} \mid i \neq j, c_{ij} \notin C$ *ou* $c_{ij} \in C, (v_i, v_j) \in R(c_{ij})\}$.

La transformation d'un CSP vers sa microstructure est réalisable en temps polynomial, la résolution du CSP se ramenant alors au fameux problème de la *Clique* [17], comme l'indique le théorème 1 [24] :

Théorème 1 *Une affectation des variables d'un CSP binaire* P *est une solution ssi cette affectation est une clique de taille* n *(le nombre de variables) dans* $\mu(P)$.

Dans la littérature, plusieurs travaux ont mis en évidence l'intérêt que recèle l'étude des microstructures pour mettre en évidence de nouvelles classes polynomiales. Elles sont parfois fondées sur la Théorie (Algorithmique) des Graphes. Ainsi, en exploitant un résultat important de Gavril [18], il est montré que si le graphe de microstructure est *triangulé* [19], le CSP peut être résolu en temps polynomial, car cela revient à résoudre le problème de la Clique, qui est NP-complet, mais de complexité linéaire pour le cas des graphes triangulés. Par la suite, Cohen a appliqué la même approche, dans [6], et a montré que les CSP dont le graphe complémentaire de la microstructure est triangulé définissent une classe polynomiale, résultat qui se déduit immédiatement de [18] et de [24].

Dans la continuité de ces travaux, Salamon et Jeavons [31] ont généralisé le résultat de [24], aux *graphes parfaits* (les graphes triangulés sont parfaits) en s'appuyant sur des résultats issus de la Conjecture de Berge sur les graphes parfaits [4, 5].

Plus récemment encore, dans [9], Cooper et al. ont introduit une classe polynomiale appelée *BTP* pour *Broken Triangle Property*. BTP permet de capter certaines classes polynomiales comme les CSP binaires acycliques. Et dernièrement, El Mouelhi et al. dans [15] ont présenté des nouveaux résultats fondés sur la microstructure, où ils montrent que des algorithmes usuels de résolution de CSP comme Backtrack (BT), Forward Checking (FC [21]) ou Real Full Look-ahead (RFL [27]) notamment, sont de complexité polynomiale sur la classe d'instances des CSP ayant une microstructure possédant un nombre polynomial de cliques maximales. Au-delà, l'étude des microstructures a également montré son intérêt dans des domaines voisins. Par exemple, pour le problème de comptage de solutions [1], ou encore l'étude des symétries dans les CSP binaires [7, 26].

Il est clair que la microstructure constitue un outil très utile pour l'étude théorique des CSP. Toutefois, cette notion n'a pu être exploitée que dans la limite des CSP binaires. Le complément de la microstructure pour le cas non binaire a été introduite par Cohen dans [6] sans pour autant aboutir à des résultats. Cette généralisation qui s'appuie sur les hypergraphes est définie de la façon suivante :

Définition 2 (Complément de la microstructure) *Étant donné un CSP* $P = (X, D, C)$*, le* complément de la microstructure *de* P *est un hypergraphe* $\overline{\mathcal{M}}(P) = (V, E)$ *avec :*
- $V = \{(x_i, v_i) : x_i \in X, v_i \in d_i\}$,
- $E = E_1 \cup E_2$ *tels que*
  - $E_1 = \{ \{(x_i, v_j), (x_i, v_{j'})\} \mid x_i \in X$ *et* $j \neq j'\}$
  - $E_2 = \{\{(x_{i_1}, v_{i_1}), \ldots (x_{i_k}, v_{i_k})\} \mid c_i \in C, S(c_i) = \{x_{i_1}, \ldots, x_{i_k}\}$ *et* $(v_{i_1}, \ldots v_{i_k}) \notin R(c_i)\}$.

Nous pouvons facilement constater que pour le cas binaire, la définition de Cohen correspond exactement au complément de la microstructure présentée dans [24]. Pour le cas non binaire, malheureusement, la notion de complément d'un hypergraphe ne semble pas avoir été étudié dans la littérature, du moins à notre connaissance. En fait, cette notion pose plusieurs questions dont la principale consiste à savoir s'il faudrait considérer toutes les hyperarêtes qui correspondent aux relations universelles, à l'image de la notion de complémentaire de graphe dans le cas binaire. Mais dans ce cas, la taille de l'hypergraphe serait potentiellement exponentielle en fonction de la taille de l'instance. Pour cette raison, vraisemblablement, cette définition n'a pas été exploitée dans la littérature. Ainsi, exploiter la définition de la microstructure basée sur les hypergraphes semble être plus difficile que l'exploitation des microstructures fondées sur les graphes.

De plus, la littérature de la Théorie des Graphes est clairement plus étendue que celle de la Théorie des Hypergraphes, avec pour conséquence de disposer ainsi d'un plus grand nombre de résultats théoriques ainsi que d'algorithmes.

Dans cette contribution, pour étendre cette notion aux CSP d'arité quelconque, nous proposons une approche différente de celle de [6], puisqu'elle s'appuie tout simplement sur la notion de graphe. A cette fin, nous présentons ici trois types possibles de microstructures basées respectivement sur les différents codages binaires des CSP non binaires : la représentation duale [10], la transformation dite par variable cachée [30] et le codage mixte. Nous étudions les propriétés de base de ces représentations, en suggérant différentes pistes pour leur exploitation théorique, notamment en vue de l'extension de classes polynomiales au cas des CSP d'arité quelconque.

Dans la section suivante, nous introduisons les différentes possibilités de microstructures pour les CSP d'arité quelconque. Ensuite, nous proposons une première exploitation de ces microstructures pour l'étude des classes polynomiales tandis que la dernière section est constituée par la conclusion de cet article.

2 Microstructures pour les CSP n-aires

La première évocation de microstructure pour le cas des CSP non binaires a été proposée par Cohen dans [6] et elle est basée sur les hypergraphes. Dans ce qui suit, nous présentons différentes alternatives à cette première approche, toutes basées sur des graphes simples. Chacune est inspirée des différents codages permettant de convertir un CSP non binaire en CSP binaire : il s'agit du codage *dual*, du codage *par variable cachée* et du codage *mixte* (parfois appelé codage *combiné*).

2.1 Microstructure basée sur le Codage Dual

Le codage dual, dans le domaine de la Programmation par Contraintes, a été employé pour la première fois dans [10]. Dans la Théorie des (Hyper)Graphes, il est appelé *Line Graph* et il est basé sur la transformation d'un hypergraphe en graphe. Dans la communauté CSP, il est appelé *Graphe Dual* et aussi *Intergraphe* dans [23], mais cette représentation avait précédemment été utilisée dans la Théorie des Bases de Données Relationnelles sous le vocable de *Qual Graphs* [2]. Dans ce codage, les variables correspondent aux contraintes du problème originel et elles sont généralement dites *variables duales*. Le domaine de chaque variable duale est exactement l'ensemble de tuples autorisés par la contrainte. Dans la représentation duale, des contraintes binaires relient deux variables duales si elles partagent au moins une variable, c'est-à-dire, si l'intersection de leurs portées n'est pas vide. Cette représentation permet de définir une représentation binaire équivalente au problème non binaire de départ, au sens où il existe une bijection entre les ensembles de solutions du CSP de départ et de celui de la représentation duale. La définition de la microstructure associée, notée *DR-Microstructure* correspond ainsi à la microstructure de ce CSP binaire équivalent :

Définition 3 (DR-Microstructure) *Étant donné un CSP $P = (X, D, C)$ d'arité quelconque, la* Microstructure basée sur la Représentation Duale *de P, dite* DR-Microstructure*, est un graphe non orienté $\mu_{DR}(P) = (V, E)$ avec :*
- $V = \{(c_i, t_i) : c_i \in C, t_i \in R(c_i)\}$,
- $E = \{\ \{(c_i, t_i), (c_j, t_j)\}\ \mid\ i \neq j, t_i[S(c_i) \cap S(c_j)] = t_j[S(c_i) \cap S(c_j)]\}$

où $t[Y]$ est la restriction de t aux variables de Y.

Notons que cette définition a été initialement introduite dans [15]. Comme pour la microstructure des CSP binaires, il existe un lien direct entre les cliques et les solutions.

Théorème 2 *Un CSP P a une solution ssi $\mu_{DR}(P)$ a une clique de taille e (le nombre de contraintes).*

Preuve : Par construction, $\mu_G(P)$ est un graphe e-parti, et toute clique contient au plus un sommet (c_i, t_i) pour chaque contrainte $c_i \in C$. Donc, une e-clique de $\mu_{DR}(P)$ correspond exactement à une clique avec un seul sommet (c_i, t_i) de chaque contrainte $c_i \in C$. De plus, chaque couple de sommets (c_i, t_i) et (c_j, t_j) reliés par une arête satisfait $t_i[S(c_i) \cap S(c_j)] = t_j[S(c_i) \cap S(c_j)]$. Donc, tous les sommets (c_i, t_i) d'une clique sont deux à deux adjacents et donc compatibles, et en particulier, une e-clique de $\mu_G(P)$ correspond exactement à e tuples t_i autorisés par toutes les contraintes, ce qui est équivalent à une solution de P. □

Considérons l'exemple suivant qui sera utilisé par la suite.

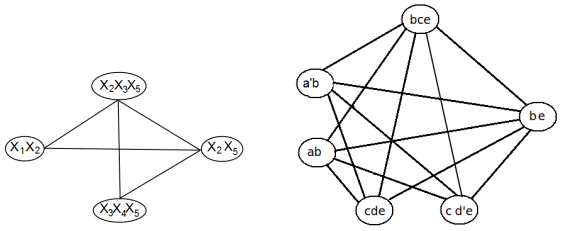
Exemple 1 *$P = (X, D, C)$ a cinq variables $X = \{x_1, \ldots, x_5\}$ avec les domaines suivants : $D = \{d_1, d_2, d_3, d_4, d_5\}$ avec $d_1 = \{a, a'\}$, $d_2 = \{b\}$, $d_3 = \{c\}$, $d_4 = \{d, d'\}$ et $d_5 = \{e\}$. $C = \{c_1, c_2, c_3, c_4\}$ est un ensemble de quatre contraintes avec $S(c_1) = \{x_1, x_2\}$, $S(c_2) = \{x_2, x_3, x_5\}$, $S(c_3) = \{x_3, x_4, x_5\}$ et $S(c_4) = \{x_2, x_5\}$. Les relations associées aux contraintes sont données par les tables suivantes :*

$R(c_1)$	
x_1	x_2
a	b
a'	b

$R(c_2)$		
x_2	x_3	x_5
b	c	e

$R(c_3)$		
x_3	x_4	x_5
c	d	e
c	d'	e

$R(c_4)$	
x_2	x_5
b	e



(a) (b)

FIGURE 1 – Graphe Dual (a) et DR-Microstructure (b) du CSP de l'exemple 1.

La DR-Microstructure de cet exemple est présentée par la figure 1. Nous avons 4 contraintes, donc $e = 4$.

Conformément au théorème 2, une solution de P est une clique de taille 4, ce qui est le cas de $\{ab, bce, be, cde\}$ Dans les exemples, nous noterons directement par t_i le sommet (c_i, t_i).

En supposant que les relations des instances sont données sous forme de tables, la taille de la DR-Microstructure est bornée par un polynôme fonction de la taille du CSP, c'est-à-dire $|E| \leq |V|^2$ avec $|V| = \Sigma_{c_i \in C} |\{t_i \in R(c_i)\}|$. De plus, il est clair que le calcul de la DR-Microstructure peut se réaliser en temps polynomial.

Il est bien connu que dans le graphe dual certaines arêtes redondantes peuvent être éliminées sans toucher à l'équivalence avec le problème de départ [22, 23]. Sur cette base, nous pouvons ainsi définir un ensemble de microstructures différentes, voisines de la DR-Microstructure. Dans [23], il est montré que pour un CSP d'arité quelconque, il existe un ensemble de CSP binaires équivalents construits sur la base de l'ensemble des intergraphes, le maximal d'entre eux correspondant au codage dual. En considérant cet ensemble de graphes partiels, nous pouvons étendre la définition précédente de DR-Microstructure :

Définition 4 (DSR-Microstructure) *Étant donné un CSP $P = (X, D, C)$ (d'arité quelconque) et un de ses intergraphes (C, F), la* Microstructure basée sur la représentation des graphes partiels duaux *de P est un graphe non orienté $\mu_{DSR}(P, (C, F)) = (V, E)$ avec :*

- $V = \{(c_i, t_i) : c_i \in C, t_i \in R(c_i)\}$,
- $E = E_1 \cup E_2$ *tels que*
  - $E_1 = \{ \{(c_i, t_i), (c_j, t_j)\} \mid \{c_i, c_j\} \in F, t_i[S(c_i) \cap S(c_j)] = t_j[S(c_i) \cap S(c_j)]\}$
  - $E_2 = \{ \{(c_i, t_i), (c_j, t_j)\} \mid \{c_i, c_j\} \notin F\}$.

Avec cette représentation, nous disposons des mêmes propriétés en termes de taille de la DSR-Microstructure puisqu'elle demeure bornée par le même polynôme que la DR-Microstructure, et le calcul de cette microstructure sera également réalisable en temps polynomial.

On peut constater que la DR-Microstructure est un graphe partiel de la DSR-Microstructure vu que pour chaque arête supprimée dans le graphe dual, une relation universelle la remplacera dans la DSR-Microstructure. Nous pouvons facilement constater que le résultat sur les cliques est toujours vrai :

Théorème 3 *Un CSP P possède une solution ssi $\mu_{DSR}(P, (C, F))$ possède une clique de taille e.*

La démonstration de ce théorème est presque identique à celle portant sur la DR-Microstructure.

2.2 Microstructure basée sur le Codage par Variable Cachée

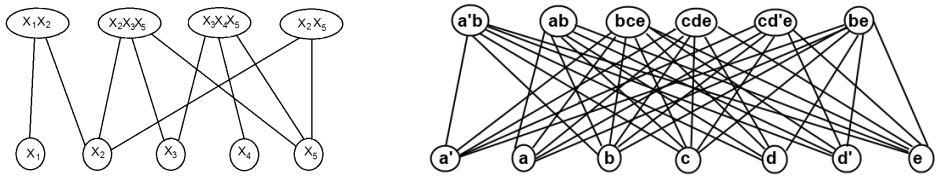
Le codage basé sur la variable cachée (*Hidden Transformation en anglais*) est inspiré par Peirce [28] (cité dans [30]). Dans cette transformation, l'ensemble de variables contient les variables originelles de X plus l'ensemble des variables duales issues de C. Les nouvelles contraintes binaires vont relier une variable duale à une variable originelle si la variable originelle appartient à la portée de la variable duale. La microstructure sera donc basée sur cette représentation binaire.

Définition 5 (HT-Microstructure) *Étant donné un CSP $P = (X, D, C)$ (d'arité quelconque), la* Microstructure basée sur la Transformation par Variable Cachée *de P est un graphe non orienté $\mu_{HT}(P) = (V, E)$ avec :*

- $V = S_1 \cup S_2$ *tel que :*
  - $S_1 = \{(x_i, v_i) : x_i \in X, v_i \in d_i\}$,
  - $S_2 = \{(c_i, t_i) : c_i \in C, t_i \in R(c_i)\}$,
- $E = \{ \{(c_i, t_i), (x_j, v_j)\} \mid$ *soit $x_j \in S(c_i)$ et $v_j = t_i[x_j]$ soit $x_j \notin S(c_i)$}.

La figure 2 représente la HT-Microstructure basée sur cette transformation pour le CSP de l'exemple 1. Nous pouvons constater que la HT-Microstructure est un graphe biparti car nous avons d'un côté les valeurs des domaines, et de l'autre, les tuples des relations. Pour rappel, une *biclique* est un sous graphe biparti complet, c'est-à-dire que chaque sommet du premier ensemble est connecté à tous les sommets du second, et réciproquement. Une biclique entre deux ensembles de sommets de taille respective i et j est notée $K_{i,j}$. Dans la HT-Microstructure, une solution correspond à une biclique particulière comme le montrera le théorème 4 qui se déduit directement des deux lemmes suivants.

Lemme 1 *Dans une HT-Microstructure, une biclique $K_{n,e}$ avec e tuples appartenant à des relations deux*



(a) (b)

FIGURE 2 – Graphe de la Hidden Transformation (a) et HT-Microstructure (b) du CSP de l'exemple 1.

à deux différentes ne peut pas contenir deux valeurs différentes d'un même domaine.

Preuve : Supposons qu'une biclique $K_{n,e}$ avec e tuples appartenant à des relations deux à deux différentes contienne deux valeurs différentes v_j et $v'_j \in d_j$. Il existe alors au moins une contrainte c_i telle que $x_j \in S(c_i)$ et par conséquent $t_i[x_j] = v_j$, v'_j ou une autre valeur v''_j. Dans les trois cas, ceci contredit que v_j et v'_j sont dans une même biclique car il n'est pas possible d'avoir deux tuples d'une même relation. □

Lemme 2 *Dans une HT-Microstructure, une biclique $K_{n,e}$ avec n valeurs issues de domaines différents ne peut pas contenir deux tuples différents d'une même relation.*

Preuve : Supposons qu'une biclique $K_{n,e}$ avec n valeurs appartenant à des variables deux à deux différentes contienne deux tuples t_i et t'_i d'une même contrainte c_i. Donc, il existe au moins une variable x_j telle que $t_i[x_j] \neq t'_i[x_j]$. Si $v_j = t_i[x_j]$ et $v'_j = t'_i[x_j]$ appartiennent toutes les deux à la biclique $K_{n,e}$, on a une contradiction car nous ne pouvons pas avoir deux valeurs d'une même variable. □

En utilisant ces deux lemmes, nous pouvons déduire que toute biclique $K_{n,e}$ avec n valeurs et e tuples tels que chaque couple de valeurs appartient à un couple de variables différentes et chaque couple de tuples appartient à un couple de relations différentes, correspond à une affectation de toutes les variables qui satisfait toutes les contraintes. Nous pouvons dans ce cas énoncer le théorème suivant :

Théorème 4 *Étant donné un CSP $P = (X, D, C)$ et sa HT-Microstructure $\mu_{HT}(P)$, P possède une solution ssi $\mu_{HT}(P)$ possède une biclique $K_{n,e}$ avec n valeurs et e tuples tels que tous les tuples appartiennent à des relations deux à deux différentes et toutes les valeurs appartiennent à des domaines deux à deux différents.*

En revenant à l'exemple précédent, nous pouvons facilement observer qu'une biclique ne correspond pas forcément à une solution. Bien que $\{a, a', b, c, e, ab, ab', bce, be\}$ soit une biclique $K_{5,4}$, elle ne constitue pas pour autant une solution. Par contre, $\{a, b, c, d, e, ab, bce, be, cde\}$ est une biclique $K_{5,4}$ et est aussi une solution de P. Donc, l'ensemble de solutions n'est pas équivalent à l'ensemble des bicliques $K_{n,e}$. Cet ensemble est équivalent seulement à l'ensemble des bicliques $K_{n,e}$ avec n valeurs et e tuples telles qu'aucun couple de valeurs (resp. de tuples) n'appartient à un même domaine (resp. à une même relation).

Comme pour la DR-Microstructure, la taille de la HT-Microstructure est bornée polynomialement par la taille du CSP :

- $|V| = \Sigma_{x_i \in X}|d_i| + \Sigma_{c_i \in C}|\{t_i \in R(c_i)\}|$ et
- $|E| \leq \Sigma_{x_i \in X}|d_i| \times \Sigma_{c_i \in C}|\{t_i \in R(c_i)\}|$.

De plus, étant donnée une instance CSP, calculer sa HT-Microstructure peut aussi se réaliser en temps polynomial.

Il existe une autre façon de représenter la microstructure à partir du codage caché et qui est liée à une autre manière pour compléter la microstructure. Nous développons ce point ci-dessous.

2.3 Microstructure basée sur le Codage Mixte

Le codage mixte [32] d'un CSP non binaire, combine à la fois le codage dual et le codage par variable caché. C'est pour cette raison que dans la littérature nous le trouvons aussi sous le nom de codage *combiné*. Cette approche consiste à connecter les valeurs des variables duales aux valeurs des variables originelles, les valeurs des variables originelles étant connectées entre elles si elles n'appartiennent pas à un même domaine et les tuples entre eux s'ils sont compatibles :

Définition 6 (ME-Microstructure) *Étant donné un CSP $P = (X, D, C)$ d'arité quelconque, la* Microstructure basée sur le codage Mixte *de P est un graphe non orienté $\mu_{ME}(P) = (V, E)$ avec :*

- $V = S_1 \cup S_2$ *tel que*
  - $S_1 = \{(c_i, t_i) : c_i \in C, t_i \in R(c_i)\}$,
  - $S_2 = \{(x_j, v_j) : x_j \in X, v_j \in d_j\}$,
- $E = E_1 \cup E_2 \cup E_3$ *tel que*
  - $E_1 = \{ \{(c_i, t_i), (c_j, t_j)\} \mid i \neq j, t_i[S(c_i) \cap S(c_j)] = t_j[S(c_i) \cap S(c_j)]\}$

- $E_2 = \{ \{(c_i, t_i), (x_j, v_j)\} \mid$ *soit* $x_j \in S(c_i)$ *et* $v_j = t_i[x_j]$ *soit* $x_j \notin S(c_i)\}$
- $E_3 = \{ \{(x_i, v_i), (x_j, v_j)\} \mid x_i \neq x_j\}$.

La microstructure basée sur le codage mixte du CSP de l'exemple 1 est donnée dans la figure 3. Nous pouvons constater que dans ce codage, nous gardons le même ensemble de sommets que celui de la HT-Microstructure. Pour les arêtes, nous aurons les arêtes de la DR-Microstructure et de la HT-Microstructure, plus toutes les arêtes reliant les valeurs de domaines différents, et correspondent à un sous-graphe n-parti complet, une partie étant alors constituée des sommets d'un même domaine. Ceci aura une influence sur les liens entre définition des solutions et les propriétés en termes de graphes, comme le précise le lemme suivant :

Lemme 3 *Dans une ME-Microstructure, une clique de $n + e$ sommets ne peut pas contenir deux valeurs d'une même variable, ni deux tuples d'une même relation.*

Preuve : Soient v_i et v'_i deux valeurs du domaine d'une variable x_i. Par définition, les sommets correspondant à v_i et v'_i ne peuvent être adjacents et donc, ne peuvent figurer ensemble dans une même clique.

De même, deux tuples d'une même relation ne sont pas compatibles entre eux et ils ne peuvent figurer ensemble dans une même clique. □

En s'appuyant sur ce lemme, nous pouvons illustrer la relation entre cliques et solutions de CSP :

Théorème 5 *Un CSP P possède une solution ssi $\mu_{ME}(P)$ possède une clique de taille $n + e$.*

Preuve : Dans la ME-Microstructure, en se référant au lemme 3, une clique de $n + e$ sommets contient exactement un sommet par variable et par contrainte. Donc, elle correspond à une affectation de n variables qui satisfait les e contraintes, et il s'agit donc d'une solution. □

Comme pour les autres microstructures, la taille de la ME-Microstructure est bornée par un polynôme fonction de la taille du CSP.

- $|V| = \Sigma_{x_i \in X}|d_i| + \Sigma_{c_i \in C}|\{t_i \in R(c_i)\}|$ et
- $|E| \leq \Sigma_{x_i \in X}|d_i| \times \Sigma_{c_i \in C}|\{t_i \in R(c_i)\}| + (\Sigma_{x_i \in X}|d_i|)^2 + (\Sigma_{c_i \in C}|\{t_i \in R(c_i)\}|)^2$.

De plus, calculer la ME-Microstructure d'un CSP est réalisable en temps polynomial.

2.4 Comparaisons entre microstructures

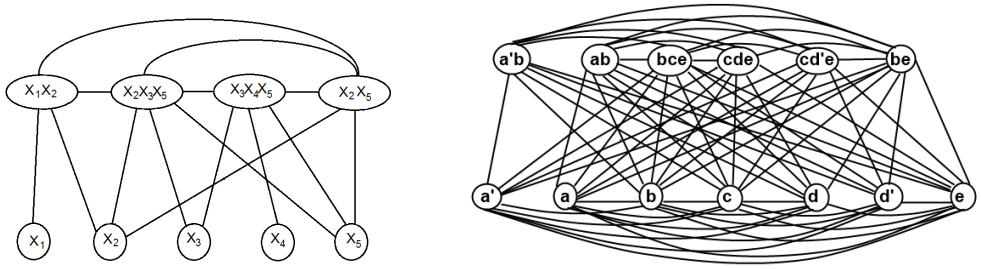
Tout d'abord, nous pouvons constater qu'aucune de ces microstructures ne peut être considérée comme une généralisation de la microstructure présentée pour le cas binaire. En effet, étant donné un CSP binaire P, nous avons $\mu(P) \neq \mu_{DR}(P), \mu(P) \neq \mu_{HT}(P)$ et $\mu(P) \neq \mu_{ME}(P)$. De plus, si la DR-Microstructure correspond exactement à la microstructure du dual, ni la HT-Microstructure, ni la ME-Microstructure ne correspondent à la microstructure du codage binaire associé, ceci étant dû à la façon dont les graphes ont été complétés. Par ailleurs, si finalement toutes ces microstructures peuvent être calculées en temps polynomial, d'un point de vue pratique, il semble difficile en général de les construire et de les manipuler efficacement, en particulier quand les contraintes ne sont pas données en extension et ceci même pour la microstructure de CSP binaires. Par contre, ce dernier point ne présente pas un obstacle pour l'étude théorique que nous désirons proposer. Il faut rappeler que notre but consiste à introduire un outil théorique qui nous permettra d'étudier les CSP d'arité quelconque. La section suivante présente quelques résultats préliminaires portant sur l'exploitation de ces représentations afin d'étendre aux CSP d'arité quelconque, des classes polynomiales définies initialement pour les seuls CSP binaires.

3 Quelques premiers résultats

Nous allons présenter maintenant quelques résultats qui peuvent être déduits de l'étude des microstructures. Pour cela, nous allons étudier trois classes polynomiales introduites pour les CSP binaires. Il s'agit de la classe des CSP ayant un nombre polynomial de cliques maximales [15], de la classe BTP [9] et enfin de la classe ZUT [8].

3.1 Microstructures et cliques maximales

Dans [15], il est montré que si le nombre de cliques maximales dans la microstructure d'un CSP binaire - noté $\omega_{\#}(\mu(P))$ - est borné par un polynôme, les algorithmes comme BT, FC ou RFL, peuvent résoudre ce CSP en temps polynomial. Plus précisément, ce coût est borné par $O(n^2 d \cdot \omega_{\#}(\mu(P)))$ pour BT et FC, et par $O(ned^2 \cdot \omega_{\#}(\mu(P)))$ pour RFL. Donc, nous montrons ici comment étendre ce résultat aux CSP d'arité quelconque en exploitant ces microstructures. Ainsi, dans [15], ces résultats ont été généralisés aux CSP non binaires en exploitant la représentation duale et en utilisant les algorithmes nBT, nFC et nRFL, soit les versions non binaires de BT, FC et RFL. Plus précisément, en exploitant un ordre particulier pour l'affectation des variables, il a été montré que la complexité est bornée par $O(nea \cdot d^a \cdot \omega_{\#}(\mu_{DR}(P)))$ pour nBT, et par $O(nea \cdot r^2 \cdot \omega_{\#}(\mu_{DR}(P)))$ pour nFC et nRFL, où a est l'arité maximale des contraintes et r le nombre



(a) (b)

FIGURE 3 – Graphe de la Transformation Mixte (a) et ME-Microstructure (b) du CSP de l'exemple 1.

maximum de tuples dans une relation. En se basant sur la complexité en temps de ces algorithmes, et en se focalisant sur les classes de graphes ayant un nombre polynomial de cliques maximales, les auteurs ont pu facilement définir de nouvelles classes traitables. Parmi ces classes de graphes, nous pouvons citer les graphes *planaires*, les graphes *Toroïdaux*, les graphes *plongeables sur des surfaces* [12] et les graphes CSG^k [3]. Le résultat peut se résumer par le théorème suivant :

Théorème 6 *Les CSP d'arité quelconque dont la DR-Microstructure est soit un graphe planaire, soit un graphe toroïdal, soit un graphe plongeable dans une surface ou soit un graphe CSG sont résolubles en temps polynomial.*

Pour la HT-Microstructure, un tel résultat ne peut pas exister. En effet, la HT-Microstructure est un graphe biparti et donc ses cliques maximales sont toutes de taille 2 puisque les cliques maximales sont limitées aux arêtes. Donc, le nombre de cliques maximales est toujours polynomial.

Pour la ME-Microstructure, un tel résultat ne peut pas plus exister, mais pour des raisons différentes. Par construction, les arêtes de l'ensemble $E_3 = \{ \{(x_i, v_i),\ (x_j, v_j)\} \mid \ x_i \neq x_j\}$ de la définition 6 permettent toutes les combinaisons de valeurs possibles. Ceci rendra le nombre de cliques maximales exponentiel à l'exception des CSP monovalent (CSP dont la taille de tous les domaines est 1).

3.2 Microstructures et BTP

La propriété BTP (Broken Triangle Property) [9] définit une classe traitable pour les CSP binaires en exploitant des caractéristiques de la microstructure. La classe BTP recèle un certain intérêt car elle capte plusieurs classes polynomiales bien connues, comme les CSP binaires arborescents ainsi que des classes polynomiales relationnelles telles que RRM. La question est donc : cette propriété peut-elle être étendue aux CSP d'arité quelconque en exploitant les caractéristiques de leurs microstructures ? Notons qu'une première évocation de cela apparaît dans [9]. Ici, nous essayons d'étendre ces travaux en utilisant les trois types de microstructures, mais auparavant, nous rappelons la propriété BTP :

Définition 7 (Broken-Triangle Property [9])
Un CSP binaire P satisfait la Broken Triangle Property (BTP) par rapport à un ordre sur les variables $<$ si, pour tout triplet de variables (x_i, x_j, x_k) tel que $x_i < x_j < x_k$, si $(v_i, v_j) \in R(c_{ij})$, $(v_i, v_k) \in R(c_{ik})$ et $(v_j, v'_k) \in R(c_{jk})$, alors soit $(v_i, v'_k) \in R(c_{ik})$, soit $(v_j, v_k) \in R(c_{jk})$. Si $(v_i, v'_k) \notin R(c_{ik})$ et $(v_j, v_k) \notin R(c_{jk})$, alors on a un triangle cassé sur x_k.

Il est prouvé dans [9] que si un CSP binaire P satisfait BTP, trouver un ordre et résoudre le CSP peut s'effectuer en $O(n^3d^4 + ed^2)$.

3.2.1 DR-Microstructure

Pour étendre BTP aux CSP non binaires, une extension de BTP, appelée DBTP (pour Dual BTP), est proposée dans [13]. La définition, qui s'appuie sur la DR-Microstructure est rappelée ci-dessous :

Définition 8 (Dual Broken-Triangle Property)
Un CSP $P = (X, D, C)$ vérifie la Dual Broken Triangle Property (DBTP) par rapport à un ordre $\prec$ sur les contraintes si pour tout triplet de contraintes (c_i, c_j, c_k) tel que $c_i \prec c_j \prec c_k$, pour tout $t_i \in R(c_i)$, $t_j \in R(c_j)$ et $t_k, t'_k \in R(c_k)$ tels que
- $t_i[S(c_i) \cap S(c_j)] = t_j[S(c_i) \cap S(c_j)]$
- $t_i[S(c_i) \cap S(c_k)] = t_k[S(c_i) \cap S(c_k)]$
- $t'_k[S(c_j) \cap S(c_k)] = t_j[S(c_j) \cap S(c_k)]$

alors
- *soit* $t'_k[S(c_i) \cap S(c_k)] = t_i[S(c_i) \cap S(c_k)]$
- *soit* $t_j[S(c_j) \cap S(c_k)] = t_k[S(c_j) \cap S(c_k)]$

La figure 4 illustre graphiquement cette définition, l'aspect formel de la définition 8 étant manifestement plus difficile à appréhender. En fait, il s'agit juste de remplacer les valeurs par les tuples et les variables par les contraintes pour avoir cette extension. Dans la figure 4 (a), nous pouvons observer la présence d'un triangle cassé sur la contrainte c_3. Donc, si on considère l'ordre suivant sur les contraintes $c_1 \prec c_2 \prec c_3$, le CSP en question ne satisfait pas DBTP par rapport à $\prec$. Au contraire, dans la figure 4(b), si t_1 et t'_3 (arête bleue) ou t_2 et t_3 (arête rouge) sont compatibles, alors le CSP satisfait DBTP par rapport à $\prec$.

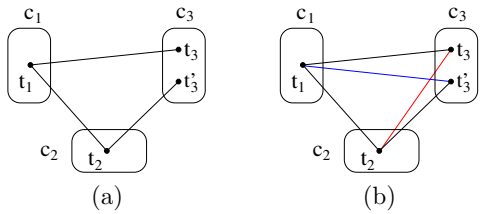


FIGURE 4 – DR-Microstructure d'un CSP n-aire satisfaisant BTP sur sa représentation duale.

Au même titre que pour BTP dont on sait que cette classe inclut les CSP binaires arborescents, dans [13] il est montré que dans le cas non binaire, les CSP dont l'hypergraphe de contraintes est β-acyclique sont aussi inclus dans DBTP. Certains autres résultats sur BTP restent vrais aussi pour DBTP. Plus de détails peuvent être trouvés dans [13].

3.2.2 HT et ME-Microstructures

Pour la HT-Microstructure, nous pouvons immédiatement constater que les triangles cassés ne peuvent pas exister car le graphe est biparti. Pour analyser BTP sur cette microstructure, on doit considérer les contraintes universelles (i.e. avec des relations universelles) entre sommets du graphe de contraintes résultant de la transformation par variable cachée. Aussi, nous étudions directement la ME-Microstructure car elle possède les mêmes sommets et par définition, elle est déjà complétée avec des arête entre ces sommets.

Considérons maintenant la HT-Microstructure. Étendre BTP est bien plus compliqué ici car nous devons considérer au moins quatre cas différents de triangles, car contrairement à BTP et DBTP sur la DR-Microstructure, nous trouvons deux types de sommets : des valeurs ou des tuples. De plus, puisque pour BTP, nous devons considérer des ordres tels que $i < j < k$, en fait, nous devons considérer six sortes de triangles sachant que les deux premiers termes de chaque triangles peuvent être permutés : (1) $x_i < x_j < x_k$, (2) $x_i < x_j < c_k$, (3) $x_i < c_j < x_k$ (ou $c_i < x_j < x_k$), (4) $x_i < c_j < c_k$ (ou $c_i < x_j < c_k$), (5) $c_i < c_j < x_k$, (6) $c_i < c_j < c_k$. Remarquons que le premier cas revient à vérifier la condition classique de BTP et que le dernier correspond à DBTP.

On peut noter l'existence d'un lien entre DR-Microstructure et ME-Microstructure pour BTP. En effet, étant donné un CSP, la présence d'un triangle cassé dans sa DR-Microstructure aura une influence sur l'ordre des contraintes. Elle aura aussi une influence sur l'ordre dans le cadre de sa ME-Microstructure, car celui-ci dépend à la fois des variables et des contraintes. Ceci nous conduit au théorème suivant :

Théorème 7 *Si un CSP P satisfait BTP en considérant sa ME-Microstructure par rapport à un ordre sur les contraintes, alors il existe un ordre sur les contraintes pour lequel P satisfait BTP en considérant sa DR-Microstructure.*

3.3 Microstructures et Classe ZUT

Dans ce qui précède, il apparaît que la DR-Microstructure semble être la microstructure la plus intéressante. Ceci restera-t-il vrai pour les autres classes polynomiales ? Pour répondre à cette question, nous allons procéder de la même façon en étudiant cette fois-ci la classe polynomiale dite *Zéro-Un-Tous* que nous noterons *ZUT* (*Zero/One/All* en anglais) introduite par Cooper et al. dans [8], et dont nous rappelons la définition :

Définition 9 (ZUT [8]) *Un CSP binaire P est dit ZUT si pour chaque contrainte c_{ij} de C, pour chaque valeur $v_i \in d_i$, c_{ij} vérifie l'une des conditions suivantes :*
- *(ZERO)* $\forall v_j \in d_j, (v_i, v_j) \notin R(c_{ij})$,
- *(UN) il existe une seule valeur* $v_j \in d_j$ *telle que* $(v_i, v_j) \in R(c_{ij})$,
- *(TOUS)* $\forall v_j \in d_j, (v_i, v_j) \in R(c_{ij})$.

Cette propriété peut être représentée graphiquement en utilisant la microstructure. Dans le cas de la DR-Microstructure, il semble facile d'appliquer le même principe que pour la microstructure classique des CSP binaires. Pour cela, par rapport à la définition de ZUT pour le cas binaire, il suffit seulement de remplacer les valeurs par des tuples. Donc, satisfaire ZUT en considérant la DR-Microstructure dépendra des relations de l'instance.

Pour la HT-Microstructure, les arêtes relient les tuples (sommets associés aux variables duales) aux valeurs (sommets associés aux variables originelles de l'instance). Nous analysons la situation selon deux directions : des tuples vers les valeurs et des valeurs vers les tuples.

– *Arêtes des tuples vers les valeurs.* Il existe deux façons pour connecter un tuple t_i à une valeur v_j. Si la variable x_j de cette valeur appartient à la portée de la contrainte c_i de ce tuple, alors t_i n'est connecté qu'à une seule valeur de x_j, à savoir v_j. Si la variable x_j n'appartient pas à la portée de la contrainte c_i de tuple t_i, alors t_i est connecté à toutes les valeurs de x_j. Donc, nous avons seulement des connexions de type « Un » ou « Tous ».
– *Arêtes des valeurs vers les tuples.* Étant donnée une contrainte associée à la HT-Microstructure, une valeur est connectée aux tuples dans lesquels elle apparaît. Nous discutons ci-dessous les différentes possibilités :
  • connexion « Zéro ». Une valeur n'est pas supportée par des tuples. C'est l'équivalent de la connexion « Zéro » de la définition de ZUT pour les CSP binaires.
  • connexion « Un ». Une valeur est supportée par un seul tuple. C'est l'équivalent de la connexion « Un » de la définition de ZUT pour les CSP binaires.
  • connexion « Tous ». Une valeur est supportée par tous les tuples de la contrainte. C'est aussi l'équivalent de la connexion « Tous » de la définition de ZUT pour les CSP binaires.

Il est donc possible qu'une instance satisfasse la propriété ZUT dans la HT-Microstructure.

Enfin, pour la ME-Microstructure, nous devons vérifier les conditions définies seulement pour la DR et HT-Microstructures car les relations entre les variables sont universelles (donc, la connexion « Tous » est toujours vraie).

Pour conclure, par construction, rien ne s'oppose à ce que les conditions de la propriété ZUT soient satisfaites, mais elles s'avèrent cependant très restrictives, comme d'ailleurs pour le cas binaire.

4 Conclusion

Dans cette contribution, nous avons introduit le concept de microstructure pour le cas des CSP d'arité quelconque. Si ce concept pour le cas binaire est désormais bien établi et utilisé comme un outil théorique, notamment pour la définition de nouvelles classes polynomiales pour les CSP, pour le cas des CSP non binaires, la notion de microstructure n'était pas clairement établie auparavant. Nous avons travaillé à la définition explicite de cette notion pour les CSP d'arité quelconque en nous basant sur différents codages binaires de CSP étudiés précédemment dans la littérature, de sorte à disposer de microstructures graphiques, plutôt qu'en nous appuyant sur la notion d'hypergraphe comme proposé dans [6]. Nous avons présenté trois types de microstructures : la DR-Microstructure (raffinée avec la DSR-Microstructure), la HT-Microstructure et la ME-microstructure, qui sont inspirées respectivement de la représentation duale, de la transformation par variable cachée et de l'approche mixte. Pour le cas binaire, aucune de ces trois microstructures ne correspond à la microstructure classique, de sorte qu'aucune d'entre elles ne peut être considérée comme une généralisation de la notion classique. Pour montrer l'intérêt de ce travail, nous avons exploité ces microstructures afin d'étendre certaines classes polynomiales définies au niveau des CSP binaires sur la base de leurs microstructures. Un premier résultat porte sur le cas des microstructures de CSP binaires dont le nombre de cliques maximales est borné par un polynôme, cas connus pour être traitables en temps polynomial par les algorithmes usuels de résolution de CSP binaires, comme BT, FC ou RFL. Ces classes s'étendent naturellement aux CSP non binaires dont les microstructures satisfont les mêmes propriétés. Nous avons également montré comment la classe BTP peut naturellement être étendue aux CSP non binaires, l'étude de cette classe étant particulièrement intéressante car elle inclut plusieurs classes polynomiales bien connus de CSP binaires, et qui par cette approche sont désormais définies en termes de contraintes d'arité quelconque.

Nous espérons que ces outils seront utilisés au niveau non binaire comme cela a pu être les cas au niveau binaire pour la microstructure classique. Par contre, il est clair qu'une utilisation pratique de ces notions semble plus que difficile, en particulier pour le cas des contraintes pour lesquelles les relations ne sont pas définies en extension. Toutefois, ces microstructures pourraient être utilisées virtuellement comme dans le cas notamment des filtrages par cohérence partielle.

Références

[1] Ola Angelsmark and Peter Jonsson. Improved algorithms for counting solutions in constraint satisfaction problems. In *CP*, pages 81–95, 2003.

[2] P.A. Bernstein and N. Goodman. The power of natural semijoins. *SIAM J. Comput.*, 10-4 :751–771, 1981.

[3] A. Chmeiss and P. Jégou. A generalization of chordal graphs and the maximum clique problem. *Information Processing Letters*, 62 :111–120, 1997.

[4] M. Chudnovsky, X. Liu G. Cornuejols, P.D. Seymour, and K. Vuskovic. Recognizing berge graphs. *Combinatorica*, 25 :143–186, 2005.

[5] M. Chudnovsky, N. Robertson, P.D. Seymour, and R.Thomas. The strong perfect graph theorem. *Ann. Math.*, 164 :51–229, 2006.

[6] David A. Cohen. A New Classs of Binary CSPs for which Arc-Constistency Is a Decision Procedure. In *CP*, volume 2833 of *LNCS*, pages 807–811. Springer, 2003.

[7] David A. Cohen, Peter Jeavons, Christopher Jefferson, Karen E. Petrie, and Barbara M. Smith. Symmetry definitions for constraint satisfaction problems. *Constraints*, 11(2-3) :115–137, 2006.

[8] M. Cooper, D. Cohen, and P. Jeavons. Characterising Tractable Constraints. *Artificial Intelligence*, 65(2) :347–361, 1994.

[9] M. Cooper, Peter Jeavons, and Andras Salamon. Generalizing constraint satisfaction on trees : hybrid tractability and variable elimination. *Artificial Intelligence*, 174 :570–584, 2010.

[10] R. Dechter and J. Pearl. Network-based heuristics for constraint satisfaction problems. *Artificial Intelligence*, 34 :1–38, 1987.

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

[12] Vida Dujmovic, Gasper Fijavz, Gwenaël Joret, Thom Sulanke, and David R. Wood. On the maximum number of cliques in a graph embedded in a surface. *European J. Combinatorics*, 32(8) :1244–1252, 2011.

[13] Achref El Mouelhi, Philippe Jégou, and Cyril Terrioux. A hybrid tractable class for non-binary csps. In *ICTAI*, pages 947–954, 2013.

[14] Achref El Mouelhi, Philippe Jégou, and Cyril Terrioux. Microstructures for csps with constraints of arbitrary arity. In *SARA*, 2013.

[15] Achref El Mouelhi, Philippe Jégou, Cyril Terrioux, and Bruno Zanuttini. Some new tractable classes of csps and their relations with backtracking algorithms. In *CPAIOR*, pages 61–76, 2013.

[16] E. Freuder. A Sufficient Condition for Backtrack-Free Search. *JACM*, 29 (1) :24–32, 1982.

[17] M.R. Garey and D.S. Johnson. *Computer and Intractability*. Freeman, 1979.

[18] F. Gavril. Algorithms for minimum coloring, maximum clique, minimum covering by cliques, and maximum independent set of a chordal graph. *SIAM Journal on Computing*, 1 (2) :180–187, 1972.

[19] M. Golumbic. *Algorithmic Graph Theory and Perfect Graphs*. Academic Press, New York, 1980.

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

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

[22] P. Janssen, P. Jégou, B. Nouguier, and M.C. Vilarem. A filtering process for general constraint satisfaction problems : achieving pairwise-consistency using an associated binary representation. In *Proceedings of IEEE Workshop on Tools for Artificial Intelligence*, pages 420–427, 1989.

[23] P. Jégou. *Contribution à l'étude des problèmes de satisfaction de contraintes : Algorithmes de propagation et de résolution – Propagation de contraintes dans les réseau dynamiques*. PhD thesis, Université des Sciences et Techniques du Languedoc, January 1991.

[24] P. Jégou. Decomposition of Domains Based on the Micro-Structure of Finite Constraint Satisfaction Problems. In *AAAI*, pages 731–736, 1993.

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

[26] Christopher Mears, Maria Garcia de la Banda, and Mark Wallace. On implementing symmetry detection. *Constraints*, 14(4) :443–477, 2009.

[27] B. Nadel. *Tree Search and Arc Consistency in Constraint-Satisfaction Algorithms*, pages 287–342. In *Search in Artificial Intelligence*. Springer-Verlag, 1988.

[28] C.S. Peirce, C. Hartshorne, and P. Weiss. *Collected Papers of Charles Sanders Peirce*, volume vol. 3. Harvard University Press, 1933.

[29] F. Rossi, P. van Beek, and T. Walsh. *Handbook of Constraint Programming*. Elsevier, 2006.

[30] Francesca Rossi, Charles J. Petrie, and Vasant Dhar. On the equivalence of constraint satisfaction problems. In *ECAI*, pages 550–556, 1990.

[31] András Salamon and Peter Jeavons. Perfect Constraints Are Tractable. In *CP*, pages 524–528, 2008.

[32] K. Stergiou and T. Walsh. Encodings of Non-Binary Constraint Satisfaction Problems. In *AAAI*, pages 163–168, 1999.

La Max-Resolution locale dans les solveurs Séparation & Evaluation pour Max-SAT

André Abramé et Djamal Habet

Aix Marseille Université, CNRS, ENSAM, Université de Toulon,
LSIS UMR 7296, 13397, Marseille
{andre.abrame, djamal.habet}@lsis.org

Résumé

L'estimation de la borne inférieure a un impact important sur les performances des solveurs de type Séparation & Evaluation pour le problème Max-SAT. A chaque nœud de l'arbre de recherche, ces solveurs détectent les sous-ensembles inconsistants (EI) de la formule par des méthodes basées sur la propagation unitaire. En fonction de la structure de ces ensembles, les solveurs les plus performants appliquent deux traitements distincts : (1) ils transforment les EI par plusieurs étapes de résolution pour Max-SAT et conservent les modifications dans le sous-arbre ou (2) ils suppriment simplement les clauses appartenant à ces EI durant l'estimation de la borne inférieure puis les réactivent avant la prochaine décision. La formule obtenue après ce dernier traitement n'est pas équivalente à l'originale et peut contenir moins d'ensembles inconsistants. Nous proposons dans ce papier une meilleure exploitation de l'ensemble des EI de la formule en les transformant systématiquement par la résolution Max-SAT, mais en ne conservant les changements que localement au nœud actuel. Les effets attendus sont une meilleure estimation de la borne inférieure et donc la réduction du nombre de décisions nécessaires pour résoudre les instances Max-SAT. Nous montrons expérimentalement l'intérêt de notre approche sur des instances Max-SAT valuées et non-valuées.

1 Introduction

Le problème Max-SAT consiste à trouver, pour une formule CNF donnée, une interprétation des variables booléennes de la formule qui maximise le nombre de clauses satisfaites. Ce problème NP-difficile [14] est la version optimisation du problème SAT. Dans la version valuée de Max-SAT, un poids positif est associé à chaque clause et l'objectif est de trouver une interprétation des variables qui maximise la somme des poids des clauses satisfaites. Il existe d'autres variantes de Max-SAT (partiel et partiel valué) qui ne sont pas traitées dans cet article (voir [1] pour plus de détails sur ces variantes).

Parmi les méthodes complètes pour résoudre Max-SAT, les solveurs les plus performants (WMAXSATZ [9, 11, 12], AKMAXSAT [7]) sont basés sur un algorithme de type Séparation & Evaluation (*Branch & Bound*, B&B). Ils explorent l'ensemble de l'espace de recherche et comparent, à chaque nœud de l'arbre de recherche, la somme des poids des clauses falsifiées par l'interprétation courante plus une sous-estimation du poids de celles qui deviendront falsifiées (la borne inférieure ou *Lower Bound*, LB) à la meilleure solution trouvée jusqu'à présent (la borne supérieure ou *Upper Bound*, UB). Si LB $\geq$ UB alors il n'existe pas de meilleures solutions dans la branche courante de l'arbre de recherche et les solveurs effectuent un retour-arrière (*backtrack*). L'estimation du nombre de clauses qui seront falsifiées par l'extension de l'interprétation courante est un élément clé pour l'efficacité des solveurs B&B : d'un coté c'est l'un des composants les plus coûteux en terme de temps d'exécution ; et de l'autre côté la qualité de l'estimation détermine le nombre de nœuds explorés. Il faut donc trouver un équilibre entre le temps passé à réaliser cette estimation et sa qualité.

Les solveurs Max-SAT de type B&B estiment la somme des poids des clauses qui seront falsifiées en comptant le nombre d'ensembles inconsistants (EI) disjoints de la formule. Ils utilisent des méthodes basées sur la propagation unitaire (*Unit Propagation*, UP) pour détecter les inconsistances (les conflits) et ils analysent les étapes de propagations ayant mené à ces conflits pour construire les sous-ensembles inconsistants. Un traitement doit être appliqué aux EI pour

s'assurer qu'ils ne soient comptés qu'une fois. Deux traitements sont utilisés par les solveurs B&B. Le premier est basé sur la règle de la résolution pour Max-SAT [2, 4, 8] qui est l'adaptation pour Max-SAT de la résolution pour SAT. Elle consiste à appliquer plusieurs étapes de Max-SAT résolution sur les clauses de l'EI. La formule résultante de cette transformation est équivalente à l'originale, mais elle peut contenir plus de clauses de plus grandes tailles. Dans un tel cas, cela peut ralentir le solveur et gêner la détection d'autres EI. Le second traitement consiste à supprimer les EI de la formule pour la durée de l'estimation de la borne inférieure. La formule obtenue n'est pas équivalente à l'originelle et peut contenir moins d'EI. Les solveurs les plus performants appliquent le traitement basé sur la Max-SAT résolution et gardent les changements dans la sous-partie de l'arbre de recherche lorsque les EI correspondent à certains modèles prédéfinis (connus sous le nom de règles d'inférence). Sinon, ils suppriment simplement les clauses des EI de la formule et ils les restaurent avant la prochaine décision.

Les travaux présentés dans cet article partent de l'hypothèse que la suppression des EI, qui est actuellement utilisée pour traiter la grande majorité des EI, donne une mauvaise estimation des inconsistances restantes dans la formule. Nous proposons de remplacer ce traitement par celui basé sur la Max-SAT résolution appliquée *localement* à chaque nœud de l'arbre de recherche, c'est à dire sans mémoriser les changements dans le sous-arbre. De cette manière, nous mémorisons la même quantité d'information que les meilleurs solveurs B&B actuels. De plus, en appliquant les modifications de manière temporaire, l'augmentation de la taille de la formule est faible et cela n'affectera pas la détection d'autres EI dans la sous-partie de l'arbre de recherche (le sous-arbre). Les bénéfices escomptés sont une amélioration de la qualité de la LB et donc la réduction du nombre de décisions nécessaires pour résoudre les instances, mais aux prix d'un traitement plus coûteux en terme de temps d'exécution. Nous avons implémenté et testé notre nouvelle méthode de traitement des EI et les résultats obtenus confirment l'intérêt de notre approche sur des instances (valuées et non-valuées) aléatoires et *crafted*.

2 Définitions et notations

Une formule valuée Φ en forme normale conjonctive (CNF) définie sur un ensemble de variables propositionnelles $X = \{x_1, \ldots, x_n\}$ est une conjonction de clauses valuées. Un clause valuée c_j est une disjonction valuée de littéraux et un littéral l est une variable x_i ou sa négation $\overline{x}_i$. On peut également représenter une formule valuée comme un ensemble de clauses valuées $\Phi = \{c_1, \ldots, c_m\}$ et une clause valuée comme un tuple $c_j = (\{l_{j_1}, \ldots, l_{j_k}\}, w_j)$ avec $\{l_{j_1}, \ldots, l_{j_k}\}$ un ensemble de littéraux et w_j un poids strictement positif. Les formules et les clauses non-valuées sont équivalentes à celles valuées avec tous les poids fixés à 1. On notera $|\Phi|$ le nombre de clauses d'une formule Φ et $|c_j|$ le nombre de littéraux d'une clause c_j.

Une interprétation φ est une application de $X' \subseteq X$ dans $\{true, false\}$. φ est complète si $X' = X$ et partielle sinon. On peut également représenter une interprétation par un ensemble I de littéraux qui ne peut pas contenir à la fois un littéral et sa négation. Si $\varphi(x_i) = vrai$ (resp. $faux$) alors $x_i \in I$ (resp. $\overline{x}_i \in I$). Un littéral l est satisfait par une interprétation I ssi $l \in I$ et il est falsifié ssi $\overline{l} \in I$. Une variable qui n'apparaît ni positivement ni négativement dans I n'est pas affectée. Une clause est satisfaite par I si au moins un de ses littéraux est satisfait et elle est falsifiée si tous ses littéraux sont falsifiés. Par convention, la clause vide (qu'on notera $\square$) est toujours falsifiée. Pour une interprétation $I = \{l\}$, on notera $\Phi|_I$ la formule obtenue en appliquant I à Φ. Formellement :

$$\Phi|_I \;=\; \{c_j \mid c_j \in \Phi, \{l, \overline{l}\} \cap c_j = \emptyset\} \cup$$
$$\{c_j / \{\overline{l}\} \mid c_j \in \Phi, \overline{l} \in c_j\}.$$

Cette notation peut être étendue à n'importe quelle interprétation $I = \{l_1, l_2, \ldots, l_k\}$ comme suit :

$$\Phi|_I \;=\; (\ldots((\Phi|_{\{l_1\}})|_{\{l_2\}})\ldots|_{\{l_k\}}).$$

Enfin, résoudre le problème Max-SAT valué consiste à trouver une interprétation complète qui maximise la somme des poids des clauses satisfaites de Φ.

3 Détection et traitement des sous-ensembles inconsistants

À chaque nœud de l'arbre de recherche, les solveurs B&B calculent la *lower bound* (LB) en additionnant la somme des poids des clauses falsifiées par l'interprétation partielle courante à une estimation de la somme des poids de celles qui seront falsifiées en étendant cette interprétation. Cette estimation est donc primordiale pour deux raisons. Premièrement, elle est réalisée très souvent et donc son temps de calcul a un impact important sur l'efficacité des solveurs. Deuxièmement, la qualité de cette estimation détermine le nombre de nœuds explorés. Il faut donc trouver un équilibre entre le temps passé au calcul de cette estimation et sa qualité. De manière simplifiée, on peut séparer cette estimation en deux partie distinctes mais étroitement liées : (1) la détection des EI disjoints et (2) leur traitement. Nous décrivons dans la suite de cette section ces deux parties.

3.1 Détection des sous-ensembles inconsistants

Nous présentons tout d'abord la méthode de détection des EI appelée propagation unitaire simulée (*Simulated Unit Propagation*, SUP) [10]. Elle consiste à appliquer la propagation unitaire (UP) à la formule à chaque nœud de l'arbre de recherche. Plus précisément, pour chaque clause unitaire $\{l\}$, SUP supprime les clauses contenant l de la formule ainsi que toute les occurrences de $\bar{l}$. Ce processus est répété jusqu'à ce qu'il n'y ait plus de clauses unitaires dans la formule ou qu'un clause vide (un conflit) soit produite. Dans ce second cas, les clauses qui ont mené par propagation unitaire au conflit forment un EI. Comme UP peut conduire à des solutions non optimale, les variables propagées sont désaffectées avant chaque nouvelle décision.

La méthode des *Failed Literals* (FL) [11] est la deuxième méthode de détection des EI la plus couramment utilisée. Considérons une formule Φ et une interprétation I. A chaque nœud de l'arbre de recherche et pour chaque variable non affectée x_i, FL applique SUP sur $\Phi|_{I\cup\{x_i\}}$ puis sur $\Phi|_{I\cup\{\overline{x}_i\}}$. Si deux EI ψ et ψ' sont détectés respectivement dans $\Phi|_{I\cup\{x_i\}}$ et $\Phi|_{I\cup\{\overline{x}_i\}}$, alors $\psi \cup \psi'$ est un EI dans $\Phi|_I$. FL est généralement appliqué lorsque SUP ne permet pas de détecter davantage d'EI.

3.2 Traitement des sous-ensembles inconsistants

Nous présentons ci-dessous les méthodes existantes pour traiter les EI détectés par SUP ou FL.

3.2.1 Suppression temporaires des EI (SEI)

Le moyen le plus simple d'éviter de détecter les mêmes EI plusieurs fois est de les supprimer de la formule. Ce traitement à l'avantage d'être rapide et la taille de la formule n'augmente pas. Comme les clauses supprimées sont restaurées avant chaque nouvelle décision, SEI n'affecte pas la qualité de la LB dans le sous-arbre. Cependant, la formule résultante de l'application de SEI n'est pas équivalente à l'originale et peut contenir moins d'EI. Les clauses appartenant aux EI sont supprimées sans considérer leurs interactions avec le reste de la formule. Cela peut augmenter le nombre de décisions nécessaires pour résoudre les instances. Dans la suite de cet article nous appelons cette méthode *suppression temporaire*.

3.2.2 La Max-SAT résolution

Une alternative pour éviter de redétecter plusieurs fois les mêmes EI est d'utiliser la règle de la Max-SAT résolution [2, 4, 8], qui est l'adaptation pour Max-SAT de la résolution pour SAT. Elle peut être définie comme suit (avec en haut la formule originale et en bas la formule obtenue après transformation) :

$$\frac{c_i = \{x, y_1, \ldots, y_s\},\ c_j = \{\overline{x}, z_1, \ldots, z_t\}}{cr = \{y_1, \ldots, y_s, z_1, \ldots, z_t\},\ cc_1,\ \ldots,\ cc_t,\ cc_{t+1},\ \ldots,\ cc_{t+s}}$$

avec :

$$\begin{gathered}
cc_1 = \{x, y_1, \ldots, y_s, \overline{z}_1, z_2, \ldots, z_t\} \\
cc_2 = \{x, y_1, \ldots, y_s, \overline{z}_2, \ldots, z_t\} \\
\vdots \\
cc_t = \{x, y_1, \ldots, y_s, \overline{z}_t\} \\
cc_{t+1} = \{\overline{x}, z_1, \ldots, z_t, \overline{y}_1, y_2, \ldots, y_s\}, \\
cc_{t+2} = \{\overline{x}, z_1, \ldots, z_t, \overline{y}_2, \ldots, y_s\}, \\
\vdots \\
cc_{t+s} = \{\overline{x}, z_1, \ldots, z_t, \overline{y}_s\}.
\end{gathered}$$

c_i et c_j sont les clauses originales, cr le résolvant et $cc_1, \ldots, cc_{t+s}$ les clauses de compensation ajoutées pour préserver le nombre de clause falsifiées par chaque interprétation (l'équivalence de la formule). On peut noter que les clauses originales c_i et c_j sont supprimées de la formule. Cette définition peut être étendue aux formules valuées comme suit : si m est le poids minimum des clauses originales, alors m est soustrait des poids de c_i et c_j et toute les clauses produites (le résolvant et les clauses de compensations) prennent le poids m.

Un EI peut être transformé en appliquant plusieurs étapes de Max-SAT résolution entre ses clauses. Ces étapes sont généralement appliquées dans l'ordre inverse des propagations. Ce traitement est proche du mécanisme d'apprentissage des clauses utilisé par les solveurs SAT modernes [13]. La Max-SAT résolution préserve l'équivalence de la formule mais elle a plusieurs désavantages. Son application est plus coûteuse en temps de calcul que celle de la suppression temporaire. La taille de la formule ainsi que la taille des clauses augmentent. À notre connaissance, tous les solveurs utilisant la Max-SAT résolution conservent les modifications dans le sous-arbre. De cette manière, les EI traités par la Max-SAT résolution sont mémorisés et cela réduit la redondance dans le calcul de la LB. Cependant, cela peut aussi avoir pour effet de limiter l'efficacité de la détection des EI en transformant des clauses qui aurait pu être utilisées par SUP ou FL. Dans le reste de ce papier, nous appelons cette méthode de traitement *Max-SAT résolution dans le sous-arbre*.

3.3 Implémentations existantes

Les solveurs Max-SAT récents, tels que WMAXSATZ [9, 11, 12] et AKMAXSAT [7], appliquent SUP et FL à chaque nœud de l'arbre de recherche pour détecter

les sous-ensembles inconsistants (EI). Si les EI correspondent à un des trois modèles définis ci-dessous, ils appliquent la Max-SAT résolution dans le sous-arbre. Sinon, ils appliquent la suppression temporaire. Ces modèles sont (avec en haut les clauses de la formule originale et en bas les clauses obtenues après transformation) :

$$(1)\frac{\{\{x,y\},\{x,\overline{y}\}\}}{\{\{x\}\}}$$

$$(2)\frac{\{\{x,y\},\{x,z\},\{\overline{y},\overline{z}\}\}}{\{\{x\},\{x,y,z\},\{\overline{x},\overline{y},\overline{z}\}\}}$$

$$(3)\frac{\{\{x_1\},\{\overline{x}_1,x_2\},\{\overline{x}_2,x_3\},\ldots,\{\overline{x}_{k-1},x_k\},\{\overline{x}_k\}\}}{\{\square,\{x_1,\overline{x}_2\},\{x_2,\overline{x}_3\},\ldots,\{x_{k-1},\overline{x}_k\}\}}$$

On peut observer que le nombre de clauses ajoutées est toujours inférieur ou égal à celui des clauses originales supprimées. Par conséquent, la taille de la formule n'augmente pas. De plus, la taille des EI correspondants à ces modèles est petite et il est donc moins probable que cela affecte la détection d'autres EI dans le sous-arbre. Toutes les clauses d'un EI ne sont pas nécessairement traitées par la même méthode. Une partie peut correspondre à un des modèles et être traité par la Max-SAT résolution tandis que l'autre partie est simplement supprimée de la formule.

Par soucis de complétude, nous décrivons également le traitement appliqué aux EI par MiniMaxSat [5, 6] ; qui est à notre connaissance le seul solveur qui fait un usage plus extensif de la Max-SAT résolution. MiniMaxSat détecte les EI par SUP, puis il applique la Max-SAT résolution dans le sous-arbre si à chaque étape de Max-SAT résolution le résolvent intermédiaire produit contient moins de 4 littéraux. Sinon, il supprime temporairement les clauses de l'EI. Comme les autres solveurs présentés ci-dessus, MiniMaxSat mémorise certains EI dans le sous-arbre tout en limitant les inconvénients de la Max-SAT résolution. Mais plutôt que de spécifier une liste de modèles, MiniMaxSat utilise un critère plus général portant sur la taille des résolvants intermédiaires. Ce critère étant moins restrictif, MiniMaxSat mémorise donc davantage d'EI mais il contrôle moins efficacement les inconvénients cités ci-dessus.

4 La Max-SAT résolution locale

Dans cette section, nous proposons une nouvelle manière de traiter les sous-ensembles inconsistants et nous présentons les grandes lignes de son implémentation.

Nous avons décrit dans les sections précédentes les méthodes existantes de traitements des EI et comment elles étaient appliquées par les solveurs Max-SAT les plus performants. Les travaux présentés dans cet article partent de l'hypothèse que la suppression temporaire, qui est appliqué lorsqu'un EI ne correspond pas aux modèles présentés précédemment, donne un estimation de la LB de mauvaise qualité. Une estimation plus précise serait donc bénéfique, même au prix d'un traitement plus coûteux en terme de temps de calcul. Nous avons mesuré (sur le benchmark utilisé en section 5) que 80% à 90% des EI ne correspondent à aucun modèle. Donc remplacer la suppression temporaire par un autre traitement induirait des changements importants dans le comportement du solveur.

Nous proposons donc de remplacer la suppression temporaire par le traitement basé sur la Max-SAT résolution, mais en restaurant les changements avant chaque nouvelle décision plutôt que de les conserver dans le sous-arbre. Ainsi, l'estimation du nombre d'inconsistances restantes dans la formule à chaque nœud de l'arbre de recherche sera plus précise. L'augmentation de la taille de la formule restera limité et cela n'affectera pas la qualité de la LB dans le sous-arbre. Nous appelons cette nouvelle méthode de traitement la *Max-SAT résolution locale*. L'exemple ci-dessous illustre les effets de chaque méthode de traitement des EI.

4.1 Illustration.

Supposons que nous sommes au niveau k de l'arbre de recherche et que nous ayons la formule non-valuée $\Phi = \{c_1,\ldots,c_{14}\}$ with $c_1 = \{x_1\}$, $c_2 = \{\overline{x}_1, x_4\}$, $c_3 = \{\overline{x}_1,\overline{x}_5\}$, $c_4 = \{\overline{x}_4, x_7\}$, $c_5 = \{x_5,\overline{x}_7\}$, $c_6 = \{x_2\}$, $c_7 = \{\overline{x}_2, x_4\}$, $c_8 = \{x_3\}$, $c_9 = \{\overline{x}_3, x_5\}$, $c_{10} = \{\overline{x}_3, x_6\}$, $c_{11} = \{\overline{x}_6,\overline{x}_7\}$, $c_{12} = \{x_2,\overline{x}_8,\overline{x}_3\}$, $c_{13} = \{\overline{x}_2,\overline{x}_8,x_9\}$ et $c_{14} = \{\overline{x}_2,\overline{x}_9\}$.

a) La suppression temporaire : L'application de SUP sur Φ (avec l'ordre de propagation UP* [11]) conduit à la séquence de propagation suivante : $< x_1@c_1, x_4@c_2, \overline{x}_5@c_3, x_7@c_5 >$ (x_1 est propagée par la clause unitaire c_1, puis x_4 par c_2, etc.). La clause c_5 est vide. Le graphe d'implication (qui décrit les étapes de propagation [13]) correspondant à cette situation est montré dans la Fig. 1(a). Le sous-ensemble inconsistant correspondant à ce conflit est $\{c_1,c_2,c_3,c_4,c_5\}$. Si on supprime cet EI de Φ, on obtient $\Phi' = \{\square, c_6,\ldots,c_{14}\}$ et toutes les propagations sont défaites.

Le traitement par SUP des clauses unitaires restantes conduit à la séquence de propagations suivante $< x_2@c_6, x_4@c_7, \overline{x}_9@c_{14}, x_3@c_8, x_5@c_9, x_6@c_{10}, \overline{x}_7@c_{11}, \overline{x}_8@c_{13} >$. Aucune clause vide n'a été détectée et il n'y a plus de clauses unitaires pour continuer la propagation (Fig. 1(b)). On peut donc aller au niveau de décision suivant $k+1$. Les clauses

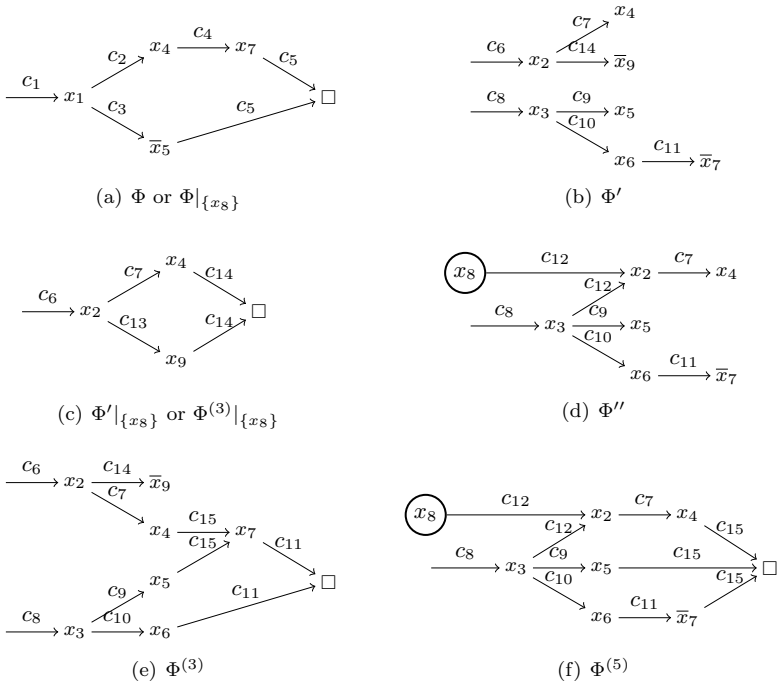


FIGURE 1 – Graphe d'implication pour chaque formule de l'exemple. Les nœuds entourés d'un cercle représentent les variables affectées par décisions tandis que les autres nœuds représentent les variables affectées par SUP. Les arcs représentent les clauses utilisées par SUP.

supprimées sont restaurées, les propagation sont défaites et une nouvelle décision $x_8 = vrai$ est faite. Comme précédemment, les variables $< x_1@c_1, x_4@c_2, \overline{x}_5@c_3, x_7@c_5 >$ sont propagées, rendant la clause c_5 vide (Fig. 1(a)), et on obtient Φ' après la suppression de l'EI correspondant. L'application de SUP sur Φ' conduit aux propagations $< x_2@c_6, x_4@c_7, x_9@c_{13} >$. La clause c_{14} est alors vide (Fig. 1(c)). L'EI correspondant $\{c_6, c_{13}, c_{14}\}$ est supprimé de la formule et on obtient $\Phi'' = \{\square, \square, c_7, \ldots, c_{12}\}$.

On continue l'application de SUP : $< x_3@c_8, x_2@c_{12}, x_5@c_9, x_6@c_{10}, x_4@c_7, \overline{x}_7@c_{11} >$. La formule ne contient plus de clause unitaire (Fig. 1(d)), et on peut passer au niveau de décision suivant. La suppression temporaire a détecté une inconsistance au niveau k et deux au niveau $k+1$

b) La Max-SAT résolution dans le sous-arbre : Comme dans l'exemple précédent, l'application de SUP sur Φ conduit aux propagations $< x_1@c_1, x_4@c_2, \overline{x}_5@c_3, x_7@c_5 >$ (Fig. 1(a)). La clause c_5 est vide et l'EI correspondant $\{c_1, c_2, c_3, c_4, c_5\}$ peut être traité. L'application de la Max-SAT résolution supprime les clauses de l'EI de la formule et ajoute une clause vide (le résolvant) et les clauses de compensations suivantes : $c_{15} = \{\overline{x}_4, \overline{x}_5, x_7\}, c_{16} = \{x_4, x_5, \overline{x}_7\}, c_{17} = \{\overline{x}_1, x_4, \overline{x}_5\}$ et $c_{18} = \{x_1, \overline{x}_4, x_5\}$. Fig. 2 montre les étapes de Max-SAT résolution appliquées durant cette transformation. On obtient la formule $\Phi^{(3)} = \{\square, c_6, \ldots, c_{18}\}$.

Les propagations sont défaites et le traitement des clauses unitaires restantes conduit aux propagations $< x_2@c_6, x_4@c_7, \overline{x}_9@c_{14}, x_3@c_8, x_5@c_9, x_6@c_{10}, x_7@c_{15} >$. La clause c_{11} est vide (Fig. 1(e)). L'application de la Max-SAT résolution supprime de la formule les clauses de l'EI $\{c_6, c_7, c_8, c_9, c_{10}, c_{11}, c_{15}\}$ et ajoute une clause vide et les clauses de compensation $c_{19} = \{x_4, \overline{x}_5, \overline{x}_6, \overline{x}_7\}$, $c_{20} = \{x_5, \overline{x}_6, \overline{x}_7\}$, $c_{21} = \{\overline{x}_4, \overline{x}_5, x_6, x_7\}$, $c_{22} = \{x_3, \overline{x}_4, \overline{x}_5, \overline{x}_6\}$, $c_{23} = \{\overline{x}_3, x_4, \overline{x}_5, x_6\}$, $c_{24} = \{\overline{x}_3, x_5, x_6\}$, $c_{25} = \{\overline{x}_3, x_4, x_5\}$, $c_{26} = \{x_3, x_4\}$ et $c_{27} = \{x_2, \overline{x}_4\}$. On obtient $\Phi^{(4)} = \{\square, \square, c_{12}, \ldots, c_{14}, c_{16}, \ldots, c_{27}\}$. Les propagations sont défaites et il ne reste plus de clauses unitaires.

Au niveau de décision $k+1$, les transformations sont conservées et une nouvelle décision $x_8 = vrai$ est réalisée. La formule ne contient pas de clauses unitaires, donc le nombre d'inconsistances reste inchangé (deux). La Max-SAT résolution dans le sous-arbre a permis de détecter deux inconsistances au niveau k et deux au

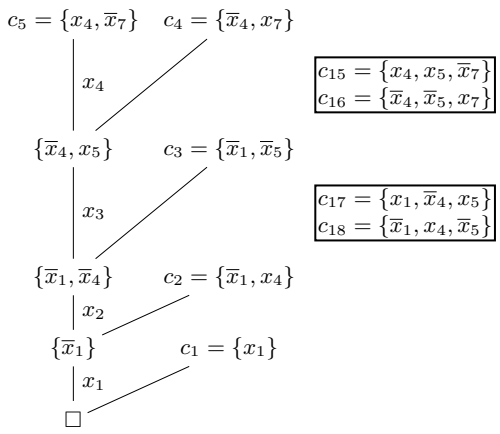


FIGURE 2 – Étapes de la Max-SAT résolution appliquées sur l'EI $\{c_1, c_2, c_3, c_4, c_5\}$ de la formule Φ. Les clauses de compensations sont encadrées.

niveau $k+1$.

c) Max-SAT résolution locale : Comme pour la Max-SAT résolution dans le sous arbre, deux inconsistances sont détectées au niveau k et après transformation on obtient la formule $\Phi^{(4)} = \{\square, \square, c_{12}, \ldots, c_{14}, c_{16}, \ldots, c_{27}\}$. Aucune propagation n'est possible.

Au niveau de décision suivant $k+1$, les propagations sont défaites et la formule originelle est restaurée. Une nouvelle décision $x_8 = vrai$ est réalisée. Les première propagations sont similaires à celles du niveau de décision précédent : $< x_1@c_1, x_4@c_2, \overline{x}_5@c_3, x_7@c_5 >$ (Fig. 1(a)). Le même conflit est détecté, qui conduit à la même transformation de la formule. Les clauses c_1, c_2, c_3, c_4, c_5 sont supprimées et $\square$, c_{15}, c_{16}, c_{17} et c_{18} sont ajouté. On obtient $\Phi^{(3)} = \{\square, c_6, \ldots, c_{18}\}$.

Les propagations sont défaites, et on peut appliquer à nouveau SUP qui conduit aux propagations suivantes : $< x_2@c_6, x_4@c_7, x_9@c_{13} >$. La clause c_{14} est vide (Fig. 1(c)). L'application de la Max-SAT résolution supprime de la formule les clauses c_6, c_{13}, c_{14} et ajoute une clause vide (aucune clause de compensation n'est ajouté). On obtient la formule $\Phi^{(5)} = \{\square, \square, c_7, \ldots, c_{12}, c_{15}, \ldots, c_{18}\}$.

À nouveau, les propagations sont défaites et on peut traiter les clauses unitaires restantes : $< x_3@c_8, x_5@c_9, x_6@c_{10}, x_2@c_{12}, \overline{x}_7@c_{11}, x_4@c_7 >$. La clause c_{15} est vide (Fig. 1(f)), et on peut appliquer la Max-SAT résolution sur l'EI correspondant $\{c_7, \ldots, c_{12}, c_{15}\}$. Les clauses originelles sont supprimées, une clause vide $\square$ est ajoutée ainsi que les clauses de compensation $c_{28} = \{x_2, \overline{x}_4, \overline{x}_5, x_7\}$, $c_{29} = \{\overline{x}_2, x_4, x_5, x_7\}$, $c_{30} = \{\overline{x}_2, x_4, \overline{x}_7\}$, $c_{31} = \{\overline{x}_2, \overline{x}_5, x_6, x_7\}$, $c_{32} = \{x_2, \overline{x}_5, \overline{x}_6, \overline{x}_7\}$, $c_{33} = \{x_5, \overline{x}_6, \overline{x}_7\}$, $c_{34} = \{\overline{x}_2, x_3, \overline{x}_5, \overline{x}_6\}$, $c_{35} = \{x_2, \overline{x}_3, x_5, \overline{x}_6\}$, $c_{36} = \{x_2, \overline{x}_3, x_6\}$ et $c_{37} = \{\overline{x}_3, x_5, x_6, \overline{x}_8\}$. On obtient $\Phi^{(6)} = \{\square, \square, \square, c_{16}, \ldots, c_{18}, c_{28}, \ldots, c_{37}\}$. Il n'y a plus de propagations possibles. La Max-SAT résolution locale permet de détecter deux inconsistances au niveau k et trois au niveau $k+1$.

Pour conclure cet exemple, la Max-SAT résolution locale a permis de détecter une inconsistance de plus que la suppression temporaire aux niveaux k et $k+1$ et une de plus que la Max-SAT résolution dans le sous-arbre au niveau $k+1$.

4.2 Implémentation

Nous avons implémenté un nouveau solveur AHMAXSAT pour étudier expérimentalement l'impact de la Max-SAT résolution locale. AHMAXSAT utilise SUP et FL pour détecter les conflits avec l'ordre de propagation UP* introduit dans [11]. Tous les EI sont traités par la Max-SAT résolution, mais la portée des changements (locale ou dans le sous-arbre) dépend de la structure des EI. Sur les clauses correspondantes aux modèles présentés précédemment, notre solveur conserve les changements dans le sous-arbre. Sur les autres clauses, les changements sont restaurés avant chaque nouvelle décision. La même portée de changement n'est pas nécessairement appliquée à toutes les clauses d'un EI.

Sans être triviale, l'implémentation de la Max-SAT résolution locale n'est pas difficile. Deux listes de changements (locale et dans le sous-arbre) sont maintenues simultanément et les interactions entre ces listes sont gérées minutieusement.

5 Etude expérimentale

Nous avons testé notre solveur AHMAXSAT sur toute les instances aléatoires et *crafted* des catégories valuées et non valuées de l'évaluation Max-SAT 2013 [1]. Nous n'avons pas inclus d'instance partielle (ou partielle valuée) ni d'instance industrielle. Même si les résultats présentés dans ce papier peuvent être naturellement étendus à ces classes d'instances, notre solveur AHMAXSAT ne les gère pas efficacement. Pour être performant sur des instances partielles, un solveur B&B doit considérer à la fois les parties dures et souples des instances. Il doit donc inclure des composants propres aux solveurs SAT, tels que l'apprentissage de clauses, une heuristique de branchement basé sur l'activité ou le retour-arrière non chronologique [3]. Notre solveur n'inclut aucun de ces composants pour le moment.

1. http://maxsat.ia.udl.cat:81.

Instances classes		#	akmaxsat		wmaxsatz2013		wmaxsatz2009		MiniMaxSat		ahmaxsat	
			S	T	S	T	S	T	S	T	S	T
unweighted	crafted/bipartite	100	100	100,3	99	297,5	99	284,0	84	811,5	100	124,7
	crafted/maxcut	67	55	345,3	55	366,7	55	401,8	55	439,4	56	346,8
	random/highgirth	82	2	1794,9	0	1800	0	1800	0	1800	5	1778,1
	random/max2sat	100	100	118,4	100	169,8	96	348,4	8	1735,1	100	127,6
	random/max3sat	100	100	193	100	242,9	97	424,2	56	1090,7	98	411,8
	random/min2sat	96	96	6,4	96	9,4	77	504,5	61	868,7	96	3,8
weighted	crafted/frb	34	14	1121,6	14	1082,9	9	1324,6	14	1116,4	14	1162,0
	crafted/ramsey	15	4	1331,1	4	1331,8	4	1342,7	4	1337,7	5	1254,3
	crafted/wmaxcut	67	64	107,2	63	176,4	61	234,5	62	241,7	61	223,1
	random/wmax2sat	120	120	50	120	134,2	119	301,5	21	1624,8	120	80,2
	random/wmax3sat	40	40	60	40	130,7	40	177,1	33	776	40	184,8
Total		821	695	348,1	691	406,6	657	551,2	398	1125,6	695	397,2

Table 1 – Comparaison des performances d'ahmaxsat avec celles des solveurs de l'état de l'art. Les deux premières colonnes donnent les classes d'instances et le nombre d'instances par classe. Pour chaque solveur, les colonnes S et T donnent respectivement le nombre d'instances résolues et le temps moyen d'exécution. Nous utilisons une pénalité de 1800 secondes pour les instances non résolues dans le calcul de T.

Pour les instances industrielles, les solveurs doivent avoir une gestion très efficace de la mémoire. À notre connaissance, aucun solveur B&B pour Max-SAT n'est capable de résoudre de très grandes instances industrielles.

Les tests présentés dans cette section ont été effectués sur des machines équipés de processeurs Intel Xeon 2,4 Ghz et de 24 Gb de mémoire vive. Le temps d'exécution est limité à 1800 secondes par instance. Nous avons testé les trois meilleurs solveurs B&B des évaluations Max-SAT 2012 et 2013 : akmaxsat, wmaxsatz2009 et wmaxsatz2013. Nous avons également inclus MiniMaxSat dans ces tests.

Les résultats détaillés (Table 1) montrent que notre solveur ahmaxsat résout autant d'instances qu'akmaxsat et quatre de plus que wmaxsatz2013. Les deux autres solveurs, wmaxsatz2009 et MiniMaxSat, ne semblent pas être compétitifs et résolvent respectivement 38 et 297 instances de moins que notre solveur. Si on considère le temps d'exécution, notre solveur est 15% plus lent que le plus performant, akmaxsat, mais 3% plus rapide que wmaxsatz2013. À nouveau, wmaxsatz2009 et MiniMaxSat sont sensiblement plus lent. On peut noter que la (relative) lenteur d'ahmaxsat comparé à akmaxsat est principalement due au instances aléatoire max-3-sat (valuées et non-valuées).

La Figure 3 compare le nombre d'instances résolues et le nombre de décisions faites par les solveurs, tandis que la Figure 4 compare pour chaque instance le nombre de décisions faites par ahmaxsat avec ceux des autres solveurs. Il est intéressant de noter que notre solveur fait moins de décisions que tout les autres solveurs [2] sur la grande majorité des instances. Comme le montre les résultats détaillés présentés dans la Table 1, la perte en temps d'exécution compense dans certain cas la réduction du nombre de décisions et le temps d'exécution de notre solveur est légèrement supérieur à celui d'akmaxsat. Même si l'application de la Max-SAT résolution ainsi que l'augmentation (limitée) du nombre de clauses de la formule à probablement un impact sur le temps d'exécution de notre solveur, il est difficile d'en déterminer l'ampleur. Les différences d'implémentation entre notre solveur et ceux de l'état de l'art rendent difficile toute comparaison précise de l'impact des différent composant sur la vitesse des solveurs.

On peut noter que nous ne présentons pas de version de notre solveur utilisant la suppression temporaire. Comme dit précédemment, le traitement appliqué sur les EI quand ils ne correspondent pas aux modèles présentés précédemment a un impact important sur le comportement du solveur. Par conséquent, l'implémentation d'une variante performante utilisant la suppression temporaire aurait requis des optimisations incompatibles avec la Max-SAT résolution locale. Plutôt que de présenter une version peu optimisée, nous avons fait le choix de comparer notre solveur aux plus performants de l'état de l'art qui utilisent la suppression temporaire. Nous pensons que cette comparaison suffit pour montrer l'intérêt de notre contribution.

6 Conclusion

Nous avons présenté dans ce papier une nouvelle méthode de traitement des sous-ensembles inconsistants qui permet une meilleure estimation de la LB et donc de réduire le nombre de nœuds examinés dans l'arbre de recherche, mais au prix d'un traitement plus coûteux en terme de temps d'exécution. Nous avons réalisé une étude expérimentale qui confirme le potentiel

2. Nous n'avons pas d'informations sur le nombre de décisions faites par MiniMaxSat, qui n'est donc pas inclus dans cette comparaison.

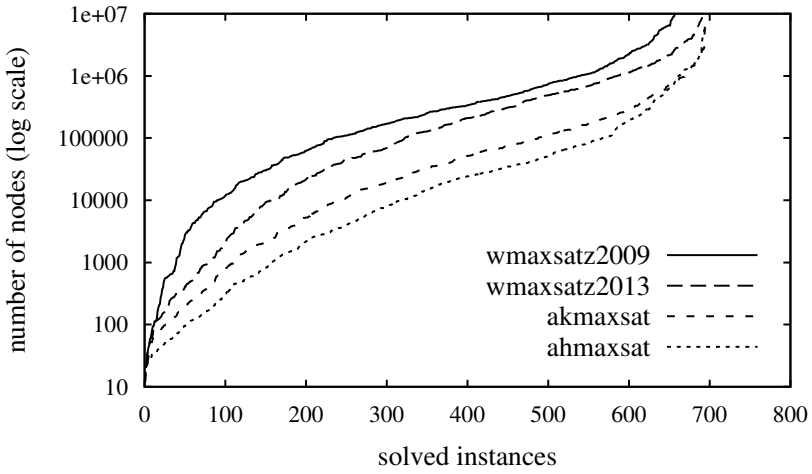


FIGURE 3 – Nombre de décisions faites par les solveurs testés.

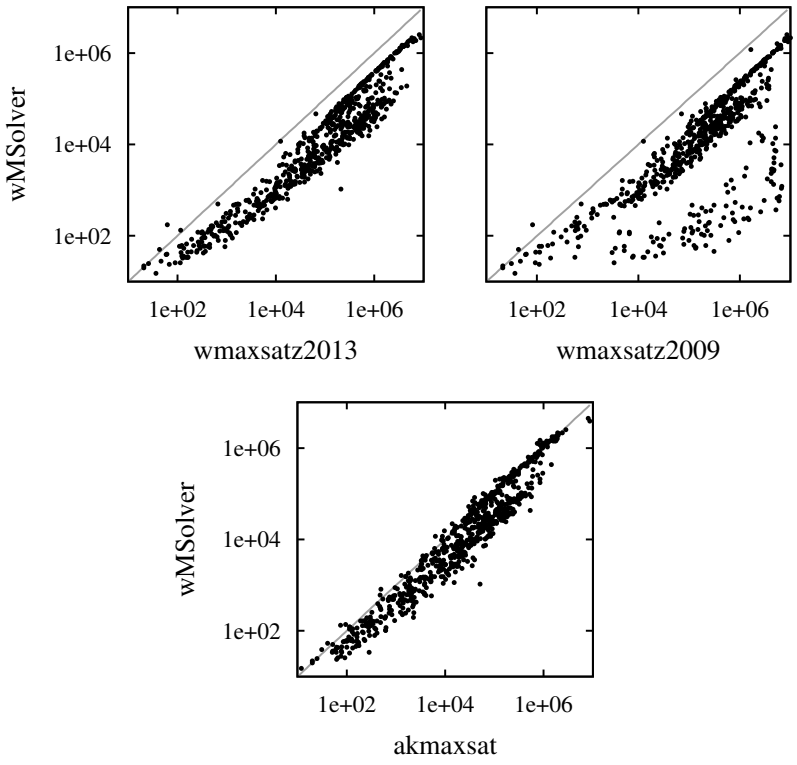


FIGURE 4 – Comparaisons du nombre de décisions faites par AHMAXSAT à ceux des autres solveurs. Chaque point représente une instance. Tous les axes sont en échelle logarithmique.

de notre approche et qui montre que notre solveur est compétitif avec les meilleurs solveurs actuels.

Dans le futur, nous essayerons d'augmenter la mémorisation réalisé par les solveurs B&B en étudiant les deux inconvénient de la Max-SAT résolution : l'augmentation de la taille de la formule et la réduction du nombre d'EI détectables par propagation unitaire. Nous étendrons également notre solveur aux instances Max-SAT partielles.

Références

[1] 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, 2009.

[2] María Luisa Bonet, Jordi Levy, and Felip Manyà. Resolution for max-sat. *Artificial Intelligence*, 171(8-9) :606–618, 2007.

[3] Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *Theory and Applications of Satisfiability Testing (SAT 2003)*, volume 2919 of *LNCS*, pages 502–518. Springer Berlin Heidelberg, 2003.

[4] Federico Heras and Javier Larrosa. New inference rules for efficient max-sat solving. In *Proceedings of the 21st national conference on Artificial intelligence - AAAI'06*, volume 1, pages 68–73. AAAI Press, 2006.

[5] Federico Heras, Javier Larrosa, and Albert Oliveras. Minimaxsat : A new weighted max-sat solver. In João Marques-Silva and Karem Sakallah, editors, *Theory and Applications of Satisfiability SAT 2007*, volume 4501 of *Lecture Notes in Computer Science*, pages 41–55. Springer Berlin / Heidelberg, 2007.

[6] Federico Heras, Javier Larrosa, and Albert Oliveras. Minimaxsat : An efficient weighted max-sat solver. *Journal of Artificial Intelligence Research - JAIR*, 31 :1–32, 2008.

[7] Adrian Kuegel. Improved exact solver for the weighted max-sat problem. In Daniel Le Berre, editor, *Pragmatics of SAT - POS'10*, volume 8 of *EPiC Series*, pages 15–27, 2012.

[8] Javier Larrosa and Federico Heras. Resolution in max-sat and its relation to local consistency in weighted csps. In Leslie Pack Kaelbling and Alessandro Saffiotti, editors, *Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence - IJCAI'05*, pages 193–198. Morgan Kaufmann Publishers Inc., 2005.

[9] Chu Min Li, Felip Manyà, Nouredine Mohamedou, and Jordi Planes. Exploiting cycle structures in max-sat. In Oliver Kullmann, editor, *Theory and Applications of Satisfiability Testing - SAT 2009*, volume 5584 of *LNCS*, pages 467–480. Springer Berlin / Heidelberg, 2009.

[10] Chu Min Li, Felip Manyà, and Jordi Planes. Exploiting unit propagation to compute lower bounds in branch and bound max-sat solvers. In Peter van Beek, editor, *Principles and Practice of Constraint Programming - CP 2005*, volume 3709 of *LNCS*, pages 403–414. Springer Berlin / Heidelberg, 2005.

[11] Chu Min Li, Felip Manyà, and Jordi Planes. Detecting disjoint inconsistent subformulas for computing lower bounds for max-sat. In *Proceedings of the 21st National Conference on Artificial Intelligence - AAAI 2006*, pages 86–91. AAAI Press, 2006.

[12] Chu Min Li, Felip Manyà, and Jordi Planes. New inference rules for max-sat. *Journal of Artificial Intelligence Research*, 30 :321–359, 2007.

[13] J. P. Marques-Silva and K. A. Sakallah. Grasp : A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5) :506–521, August 1999.

[14] Christos H. Papadimitriou. *Computational complexity*. Addison-Wesley, 1994.

Contrainte de non-chevauchement entre objets décrits par des inégalités non-linéaires

Ignacio Salas[1] **Gilles Chabert**[1] **Alexandre Goldsztejn**[2]

[1] Mines de Nantes - LINA (UMR 6241)
[2] CNRS - LINA (UMR 6241)
{ignacio.salas,gilles.chabert}@mines-nantes.com alexandre.goldsztejn@univ-nantes.fr

Résumé

Le placement d'objets 2D dans un espace limité est un problème omniprésent aussi bien sur le plan académique qu'industriel. Quel que soit le contexte, la résolution de ce problème exige la capacité de pouvoir déterminer où un premier objet peut être placé de telle façon qu'il ne chevauche pas un second objet, précédemment placé. Ce sous-problème s'appelle la contrainte de non-chevauchement. La complexité de cette contrainte de non-chevauchement dépend du type d'objets considérés. Elle est simple dans le cas de rectangles. Elle a également été étudiée dans le cas de polygones. Cet article propose une approche numérique pour la classe générale des objets décrits par des inégalités non-linéaires. Notre objectif ici est de *calculer* la contrainte de non-chevauchement, c'est à dire, de décrire l'ensemble de toutes les positions et orientations qui peuvent être attribuées au premier objet de telle sorte que l'intersection avec le second soit vide. Nous nous basons sur un algorithme de branch & prune dédié. Nous montrons d'abord que la contrainte de non-chevauchement équivaut à une somme de Minkowski, même lorsque l'orientation est prise en compte. Nous en déduisons un *contracteur intérieur*, c'est à dire, un opérateur qui élimine du domaine courant un sous-ensemble de positions et orientations qui violent nécessairement la contrainte de non-chevauchement. Ce contracteur intérieur est intégré dans une boucle de *sweep*, une technique utilisée jusqu'ici uniquement pour les domaines discrets. Nous aboutissons ainsi à un algorithme de branch & prune présentant de bien meilleures performances que Rsolver, outil de référence pour la résolution de contraintes quantifiées en domaines continus.

1 Introduction

L'objectif de cet article est de calculer l'ensemble de toutes les positions et orientations qui peuvent être données à un objet de telle sorte qu'il ne chevauche pas un second objet (voir la figure 1, qui montre le cas plus simple où l'orientation n'est pas considérée). Nous abordons le cas général d'objets décrits par des inégalités non-linéaires. Calculer cet ensemble est une tâche centrale pour la résolution de problèmes de placement, qui consistent à placer un ensemble d'objets dans un espace limité de telle sorte qu'ils ne se chevauchent pas deux à deux.

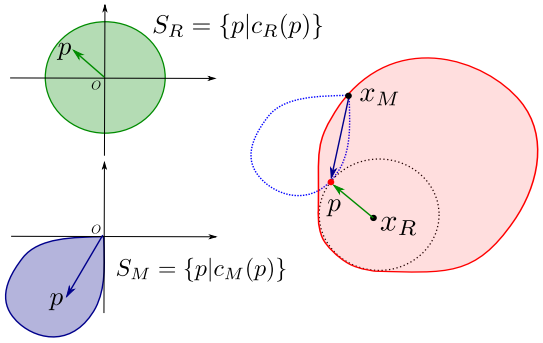


FIGURE 1 – **Contrainte de non-chevauchement**. *Gauche :* deux objets S_R et S_M. *Droite :* la région rouge représente l'ensemble de toutes les positions x_M pour S_M où la contrainte de non-chevauchement est violée.

Dans cette introduction, nous définissons d'abord ce qu'est un *objet* dans notre contexte et formalisons la contrainte de non-chevauchement. Nous explicitons ensuite le type d'objets considéré puis expliquons ce que signifie ici *calculer* un ensemble. Nous mentionnons enfin d'autres travaux liés à cette problématique.

Dans la section 2, nous montrons que notre prob-

lème équivaut au calcul d'une somme de Minkowski. Sur la base de cette observation, nous proposons un algorithme de branch & prune dans la section 3. Des résultats expérimentaux sont donnés dans la section 4, avant de conclure.

1.1 Définition des objets

Par simplicité, supposons d'abord que l'orientation est fixée, c'est à dire, que les objets peuvent seulement être translatés.

Déplacer un objet revient à fixer la position d'un point particulier que nous appelons *l'origine*. Cette origine peut être choisie arbitrairement. Par exemple, dans le placement de rectangles, l'origine d'un rectangle peut être un sommet ou le milieu. Une fois cette convention faite pour l'origine, la forme de l'objet est simplement une contrainte usuelle. Pour l'illustrer, considérons de nouveau le placement de rectangles. Si son origine est le coin inférieur-gauche, le rectangle de dimensions l_1 et l_2 est l'ensemble de tous les $p \in \mathbb{R}^2$ tels que

$$c(p) \iff 0 \leq p_1 \leq l_1 \wedge 0 \leq p_2 \leq l_2. \quad (1)$$

La contrainte

$$c(p) \iff -\frac{l_1}{2} \leq p_1 \leq \frac{l_1}{2} \wedge -\frac{l_2}{2} \leq p_2 \leq \frac{l_2}{2}. \quad (2)$$

correspond, elle, à un rectangle dont l'origine est le milieu. Cette contrainte, bien sûr, n'est qu'un simple décalage de la contrainte précédente. Ainsi, la forme d'un objet peut être exprimée par une simple contrainte, moyennant une convention implicite pour l'origine. Prenons un dernier exemple : un cercle de rayon r est l'ensemble de tous les points $p \in \mathbb{R}^2$ de tels que

$$c(p) \iff \|p\| \leq r, \quad (3)$$

l'origine étant, dans ce cas, le centre du cercle.

La contrainte $c(p)$ définit un objet sans déplacement ni rotation. La contrainte plus générale définissant un objet placé en x et tournée d'un angle α est facilement obtenue comme suit. Tout d'abord, dans le cas d'une simple translation, la partie du plan occupée par l'objet placé en x est l'ensemble de tous les points p tels que $c(p - x)$ est satisfait. Ajoutons maintenant l'orientation. Avec un argument géométrique classique, un objet placé en x et tournée d'un angle α est la contrainte

$$c\big(R_{-\alpha}(p - x)\big) \quad (4)$$

où R_α est la matrice de rotation d'angle α :

$$R_\alpha = \begin{pmatrix} \cos(\alpha) & -\sin(\alpha) \\ \sin(\alpha) & \cos(\alpha) \end{pmatrix}. \quad (5)$$

De façon équivalente, nous pouvons dire que $c(p) \equiv c(R_0(p - 0))$ représente l'objet placé en 0 et tournée d'un angle 0. Pour conclure :

- *un objet* est une contrainte dans le plan (c.a.d., avec deux variables),
- *placer* un objet signifie fixer les coordonnées de son origine implicite,
- *orienter* un objet signifie fixer l'angle de rotation de son origine implicite.

Dans cet article nous considérons des objets décrits par des inégalités non-linéaires $c(p) \iff f(p) \leq 0$. Ex., un cercle de rayon 1 dont l'origine est le milieu est l'ensemble de tous les $p \in \mathbb{R}^2$ tels que $f(p) \leq 0$ avec $f : p \mapsto \|p\| - 1$. Pour clarifier la présentation, nous supposons que chaque objet est décrit par une seule inégalité, mais nos résultats peuvent être étendus facilement au cas plus général des objets décrits par des formules logiques du premier ordre (disjonctions de conjonctions d'inégalités). Il n'est également émis aucune hypothèse sur les fonctions impliquées, hormis le fait qu'elles soient définies par des expressions mathématiques basés sur les opérateurs standards ($+$, $\times$, $\sqrt{\cdot}$, exp, etc.). En particulier, il n'y a aucune hypothèse de convexité sur les objets en entrée.

1.2 Contrainte de non-chevauchement

Nous pouvons nous concentrer maintenant sur la contrainte de non-chevauchement. La contrainte de non-chevauchement implique deux objets, l'un étant fixé, l'autre représentant les inconnues du problème. Pour cette raison, nous appellerons "objet de référence" le premier objet et noterons c_R la contrainte le décrivant. Le deuxième objet sera appelé "objet mobile" et sa contrainte notée c_M.

Nous allons montrer à la fin de cette section que le cas général où les deux objets sont translatés et/ou tournés peut en fait être obtenu à partir du cas plus simple où l'objet de référence n'a pas subi de transformation. Intuitivement, le repère dans lequel la contrainte de non-chevauchement est établie peut être centré sur l'objet de référence et aligné avec son orientation, quoique la formule exacte ne soit pas si triviale.

Nous allons donc introduire dans la définition ci-dessous uniquement la position x_M et l'angle de rotation α_M de l'objet mobile. La contrainte de non-chevauchement est la négation de la contrainte de chevauchement qui peut être déclarée ainsi :

Définition 1 (Contrainte de chevauchement) *Étant donné deux contraintes c_R et c_M, un vecteur $x_R \in \mathbb{R}^2$ et $\alpha_R \in [0, 2\pi]$:*

$$\begin{gathered}\text{overlap}_{(c_R,c_M)}(x_M, \alpha_M) \iff \\ \exists p \in \mathbb{R}^2, c_R(p) \wedge c_M\big(R_{-\alpha_M}(p - x_M)\big).\end{gathered} \quad (6)$$

Dans le cas d'une translation uniquement, cela ce simplifie en

$$\begin{array}{c}\text{overlap}_{(c_R,c_M)}(x_M) \iff \\ \exists p \in \mathbb{R}^2 c_R(p) \wedge c_M(p - x_M).\end{array} \quad (7)$$

Notre objectif est de *calculer* la contrainte de chevauchement. *Calculer* signifie qu'une représentation (numérique mais garantie) explicite de l'ensemble solution $\mathcal{S}' := \{(x_M, \alpha_M),\ \text{overlap}_{(c_R,c_M}(x_M, \alpha_M)\}$ (ou $\mathcal{S} := \{x_M,\ \text{overlap}_{(c_R,c_M)}(x_M)\}$ dans le cas du déplacement) doit être renvoyée par notre algorithme.

Nous montrons maintenant que la satisfiabilité de la contrainte de chevauchement dans le cas où l'objet de référence possède une position x_R et une orientation α_R peut être testée en utilisant $\mathcal{S}'$ (et notamment sa représentation explicite). Plus précisément :

Proposition 1 *L'objet mobile de position x_M et d'orientation α_M chevauche l'objet de référence de position x_R et d'orientation α_R ssi*

$$(R_{-\alpha_R}(x_M - x_R), \alpha_M - \alpha_R) \in S' \quad (8)$$

Preuve 1 *Par définition, (x_M, α_M) satisfait la contrainte de chevauchement avec l'objet de référence placé en x_R et tourné de α_R ssi il existe $\exists p \in \mathbb{R}^2$ tel que $c_R(R_{-\alpha_R}(p - x_R))$ et $c_M(R_{-\alpha_M}(p - x_M))$. Cela équivaut à ce qu'il existe $q \in \mathbb{R}^2$ obtenu ainsi :*

$$q = R_{-\alpha_R}(p - x_R) \iff p = R_{\alpha_R} q + x_R, \quad (9)$$

tel que $c_R(q)$ et

$$\begin{array}{c}c_M(R_{-\alpha_M}(R_{\alpha_R} q + x_R - x_M)) \iff \\ c_M(R_{-\alpha_M+\alpha_R}(q + R_{-\alpha_R}(x_R - x_M)));\end{array} \quad (10)$$

ce qui équivaut à (8). ▲

Remarque 1 *Les applications de placement considèrent la contrainte de non-chevauchement (plutôt que la contrainte de chevauchement). La caractérisation de cette dernière est obtenue en échangeant simplement les rôles des deux ensembles $\mathcal{I}$ et $\mathcal{O}$ dans la définition 2 ci-dessousb. La contrainte de chevauchement est préférée ici car elle simplifie les expressions des contraintes et leur lien avec la somme de Minkowski, présentée dans la section 2.*

1.3 Intervalles, Boîtes et Pavage

La représentation que nous utilisons est un *pavage*. Cette représentation est un choix naturel dans le contexte de la programmation par contraintes où la grande majorité des algorithmes sur les variables continues, pour ne pas dire tous, supposent des domaines sous forme d'intervalles (voir, e.g., [3, 8]). Dans la définition ci-après, nous appelons *boîte* un produit cartésien de d intervalles où d est 2 dans le cas de $\mathcal{S}$ et 3 dans le cas de $\mathcal{S}'$.

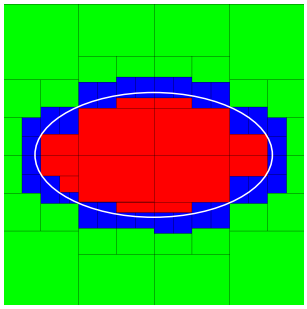


FIGURE 2 – **Pavage d'une ellipse**. Les boîtes rouges à l'intérieur appartiennent à $\mathcal{I}$, les boîtes bleues à la frontière appartiennent à $\mathcal{B}$ et les boîtes vertes à l'extérieur appartiennent à $\mathcal{O}$.

Définition 2 (Pavage) *Un pavage d'un ensemble $\mathcal{S} \subset \mathbb{R}^d$ est un triplet $(\mathcal{I}, \mathcal{B}, \mathcal{O})$ où $\mathcal{I}$ (pour "intérieure'), $\mathcal{O}$ (pour "extérieur") et $\mathcal{B}$ (pour "bord") sont trois ensembles de boîtes vérifiant*

$$\cup\mathcal{I} \subset \mathcal{S}, \quad (\cup\mathcal{O}) \cap \mathcal{S} = \emptyset \quad et \quad \cup(\mathcal{B} \cup \mathcal{I} \cup \mathcal{O}) = \mathbb{R}^d. \quad (11)$$

Un exemple de pavage apparaît dans la figure 2.

1.4 Contribution et Travaux Liées

Cet article propose une algorithme calculant un pavage de la contrainte de chevauchement. Une première contribution est sur la modélisation du problème : Nous montrons que la contrainte de chevauchement peut être exprimée comme une somme de Minkowski. D'un côté, cela généralise l'approche et simplifie la description de l'algorithme. De l'autre côté, cela permet de traiter de manière homogène le cas simple de la translation et le cas plus complexe combinant translation et rotation, en remplaçant l'angle de rotation par une cordonnée de translation supplémentaire dans un espace "augmenté" (voir proposition 2). L'autre contribution est algorithmique. Nous proposons un contracteur *intérieur* original pour ce problème, c'est à dire, un opérateur identifiant une partie de l'ensemble solution. Cet opérateur met en œuvre une *boucle de sweep* et exploite les propriétés de la somme de Minkowski. Le deuxième opérateur est un test de rejet extérieur basé sur une propagation de contraintes classique. Les deux opérateurs sont entrelacés dans un algorithme de branch & prune, qui calcule le pavage souhaité.

D'un autre point de vue, la définition 1 signifie que notre problème appartient à la catégorie des contraintes existentiellement quantifiées. Un algorithme

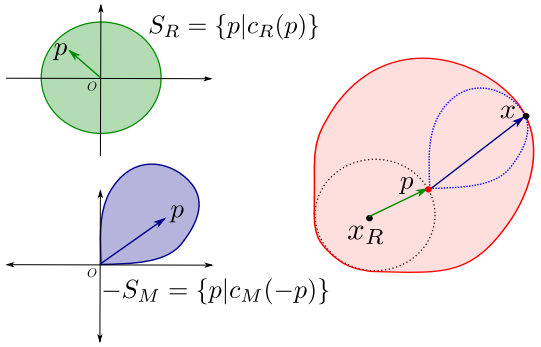


FIGURE 3 – **Somme de Minkowski.** La somme de Minkowski de S_R (objet de référence) et $-S_M$ (symétrique de l'objet mobile) coïncide avec la description de la contrainte de non-chevauchement dans la figure 1.

de l'état de l'art pour calculer le pavage d'inégalités existentiellement quantifiées est décrit dans [11] et mis en œuvre dans l'outil RSOLVER [10] (Rsolver est un algorithme générique, sorte de variante numérique de CAD, pour résoudre des contraintes quantifiées). Notons que des techniques générales et efficaces pour les contraintes d'égalité existent aussi, cf. [6, 7].

Finalement, mentionnons qu'une formule exacte pour la contrainte de chevauchement $\mathcal{S}$ a été donnée dans [2] pour le cas où les objets sont des polytopes. L'ensemble, dans ce cas, est l'enveloppe convexe des points obtenus en sommant un sommet du premier polytope avec un sommet du second.

2 Contrainte de chevauchement et somme de Minkowski

Dans cette section, nous montrons que la contrainte de chevauchement peut être reformulée comme une somme de Minkowski. Cette reformulation sous-tend le solveur branch & prune que nous présentons plus loin. Rappelons d'abord la définition de la somme de Minkowski de deux ensembles :

Définition 3 (Somme de Minkowski)
Étant donné deux ensembles équi-dimensionnels $S_1, S_2 \subseteq \mathbb{R}^d$, la somme de Minkowski est

$$S_1 + S_2 = \{x_1 + x_2 \in \mathbb{R}^d : \ x_1 \in S_1, \ x_2 \in S_2\}. \quad (12)$$

La différence de Minkowski est définie en conséquence par :

$$S_1 - S_2 = \{x_1 - x_2 \in \mathbb{R}^d : \ x_1 \in S_1, \ x_2 \in S_2\}. \quad (13)$$

La figure 3 montre un exemple de deux ensembles et leur somme de Minkowski.

En considérant S_1 comme un contrainte c_1 (i.e., $x \in S_1 \iff c_1(x)$) et de façon similaire, S_2 comme la contrainte c_2, nous avons, de façon équivalente :

$$S_1 + S_2 = \{x \in \mathbb{R}^d : \ \exists p \in \mathbb{R}^d, \ c_1(p) \wedge c_2(x-p)\}, \quad (14)$$

où d est le nombre de variables dans la contrainte. En comparant (14) et (7), on voit immédiatement que $\mathcal{S} = S_R - S_M$, c.a.d., la contrainte de chevauchement peut être représentée comme une somme de Minkowski dans le cas d'une translation seulement.

Nous montrons maintenant que $\mathcal{S}'$ est aussi une somme de Minkowski, un résultat moins trivial. Pour cela, nous plongeons l'objet mobile $S_M \subseteq \mathbb{R}^2$ dans $\mathbb{R}^3$ en encondant la rotation dans la dimension supplémentaire :

$$\begin{aligned} \mathcal{S}'_M &:= \{(v, \beta) : c_M(R_\beta \, v)\} \\ &= \{(v, \beta) : R_\beta \, v \in \mathcal{S}_M\}. \end{aligned} \quad (15)$$

La proposition suivante énonce alors que la contrainte de chevauchement avec rotation $\mathcal{S}'$ peut être réécrite comme une différence de Minkowski entre deux tels ensembles "augmentés" (voir figure 4).

Proposition 2

$$\mathcal{S}' = S_R \times \{0\} - S'_M. \quad (16)$$

Preuve 2 *Par définition, $(x_M, \alpha_M) \in \mathcal{S}'$ est vrai si et seulement si $\exists p \in \mathbb{R}^2$ tel que $c_R(p)$ et $c_M(R_{-\alpha_M}(p - x_M))$.*

De façon équivalente, il existe $u_R \in \mathcal{S}_R$ et $u_M \in \mathcal{S}_M$ tels que $u_R = p$ et $u_M = R_{-\alpha_M}(u_R - x_M) \iff x_M = u_R - R_{\alpha_M} u_M$. Finalement le vecteur (x_M, α_M) est prouvé comme étant la somme de $(u_R, 0) \in \mathcal{S}_R \times \{0\}$ et $(R_{\alpha_M} u_M, \alpha_M) \in \mathcal{S}'_M$. ▲

3 Algorithme

Notre objectif est désormais de calculer un pavage $(\mathcal{I}, \mathcal{B}, \mathcal{O})$ de la somme $\mathcal{S}$ de deux ensembles $\mathcal{S}_1$ et $\mathcal{S}_2$. D'après la section précédente, le lien avec la contrainte de non-chevauchement se fait en remplaçant $\mathcal{S}_1$ par S_R ou S'_R et $\mathcal{S}_2$ par $-S_M$ ou $-S'_M$.

Notre algorithme est basé sur un branch & bound récursif classique de type SIVIA [9, 5]. L'opération centrale effectuée sur chaque boîte $[x]$ se divise en trois parties. D'abord, $[x]$ est contracté en une boîte $[x]'$ par un *contracteur intérieur*, c'est à dire, un opérateur qui garantit :

$$[x]' \subseteq [x] \ \wedge \ [x] \backslash [x]' \subseteq \mathcal{I}. \quad (17)$$

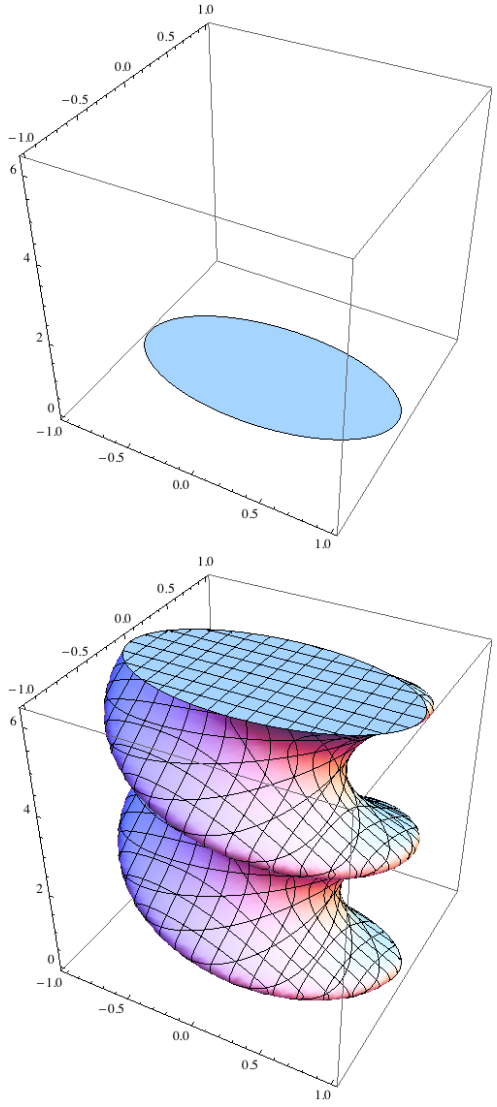


FIGURE 4 – Ensembles dont la différence de Minkowski donne la contrainte de chevauchement de deux ellipses, lorsque la rotation est prise en compte. *Gauche :* l'ellipse de référence augmentée, avec la coordonnée de rotation mise à 0. *Droite :* l'ellipse mobile augmentée S'_M.

Si $[x]' \neq \emptyset$ alors $[x]'$ est contracté en une boîte $[x]''$ par un *contracteur extérieur* qui garantit :

$$[x]'' \subseteq [x]' \ \wedge \ [x]' \backslash [x]'' \subseteq \mathcal{O}. \tag{18}$$

Finalement, si $[x]'' \neq \emptyset$ alors $[x]''$ est découpé en deux nouvelles boîtes qui sont ajoutées dans la liste des boîtes frontières $\mathcal{B}$. La récursivité s'arrête quand la surface totale des boîtes dans $\mathcal{B}$ est en dessous de $\varepsilon\%$ de la surface de la boîte initiale, où ε est un paramètre défini par l'utilisateur.

L'originalité de notre approche réside dans le contracteur intérieur, que nous allons décrire maintenant.

3.1 Contracteur Intérieur

Notre contracteur intérieur est basé sur *l'arithmétique intérieure*, une variante de l'arithmétique d'intervalles classique qui permet de construire une sous-boîte d'une boîte $[x]$, à l'intérieur d'un ensemble donné $\mathcal{S}$ décrit par des inégalités. Cette technique a été introduite pour la première fois dans §3 de [4] puis utilisée dans [1] dans le cadre de l'optimisation globale. Cette arithmétique intérieure peut aussi être utilisée avec un point initial (ou une boîte initiale) qui est alors *étendu*. Plus précisément, étant donné une boîte $[x]$ et $\tilde{x} \in [x]$, elle produit une boîte $[\tilde{x}]$ vérifiant

$$\tilde{x} \in [\tilde{x}] \subseteq [x] \ \wedge \ [\tilde{x}] \subseteq \mathcal{S}, \tag{19}$$

ou une boîte vide si $\tilde{x} \notin \mathcal{S}$. Cette arithmétique partage des propriétés similaires avec son pendant classique : la complexité temporelle est en la longueur de l'expression de la contrainte et elle produit une boîte optimale $[\tilde{x}]$ (c.a.d., de taille maximale sur chaque dimension) si aucune variable n'apparaît deux fois dans l'expression.

Avant de décrire la façon dont une boîte $[x]$ peut être contractée avec cette nouvelle arithmétique, adressons d'abord une question plus simple : comment trouver une sous-boîte de $[x]$ qui soit à l'intérieure de $\mathcal{S}$?

Une réponse possible consiste à chercher deux boîtes $[x]_1$ et $[x]_2$ telles que

$$[x]_1 \subseteq \mathcal{S}_1, \quad [x]_2 \subseteq \mathcal{S}_2 \ \text{ et } \ ([x]_1 + [x]_2) \cap [x] \neq \emptyset \tag{20}$$

car, dans ce cas, $([x]_1 + [x]_2) \subseteq [x] \cap \mathcal{S}$. Pour trouver de telles boîtes, nous pouvons calculer en parallèle deux pavages, le premier pour $\mathcal{S}_1$, l'autre pour $\mathcal{S}_2$, et arrêter le processus lorsque deux boîtes qui satisfont (20) sont identifiées. En combinant les boîtes du premier pavage avec celles du deuxième, cette approche revient à exécuter un branch & bound dans un espace $(2 \times d)$-dimensionnel. Notons que ce branch & bound est un sous-solveur embarqué dans le solveur principal, celui pour la variable x. Notre objectif est de réduire le sous-solveur à seulement d dimensions, ce qui, en passant, est le coût incompressible à payer pour manipuler d paramètres existentiellement quantifiés.

À cette fin, nous utilisons la même idée que ci-dessus, mais en se basant cette fois sur la relation (14) (voir figure 5). Pour construire une boîte intérieure dans $[x]$, nous supposons tout d'abord qu'un point quelconque $\tilde{x}$ a été choisi à l'intérieur de $[x]$ (ce point est, en fait, produit automatiquement par la boucle

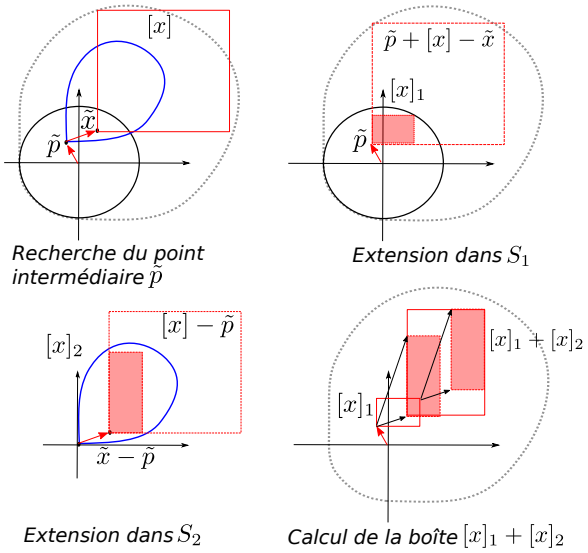


FIGURE 5 – **Étapes du contracteur intérieur.**

de *sweep*, comme cela sera expliqué dans la figure 6). Nous cherchons alors un autre point $\tilde{p}$ tel que

$$c_1(\tilde{p}) \wedge c_2(\tilde{x}-\tilde{p}). \tag{21}$$

La recherche de ce point est la tâche du sous-solveur [1].

Lorsque que $\tilde{p}$ est trouvé, il est "étendu" à une sous-boîte $[x]_1$ de $(\tilde{p}+[x]-\tilde{x})$ à l'intérieur de $\mathcal{S}_1$ grâce à l'arithmétique intérieure. Le point $(\tilde{x}-\tilde{p})$ est également étendu à une sous-boîte $[x]_2$ de $([x]-\tilde{p})$ a l'intérieur de $\mathcal{S}_2$. Cependant, si $\tilde{x}$ s'avère être à l'extérieur de $\mathcal{S}$, cette dernière inflation échoue et le processus, dans ce cas, est arrêté. Sinon, la boîte qui résulte satisfait $([x]_1+[x]_2) \subseteq \mathcal{S}$ et $([x]_1+[x]_2) \cap [x] \neq \emptyset$.

Notons que $([x]_1+[x]_2) \cap [x] \neq \emptyset$ est seulement une conséquence de $\tilde{x} \in [x]$. Donc les boîtes initiales utilisées pour les deux extensions sont en quelque sorte arbitraires, mais la façon dont nous les avons fixées est un heuristique qui tend à maximiser la surface de la boîte finale $([x]_1+[x]_2) \cap [x]$.

Maintenant que nous possédons une technique pour construire une boîte intérieure dans $[x]$ contenant un point spécifique $\tilde{x}$, nous pouvons utiliser ce service dans une boucle de *sweep*. La boucle de sweep peut être vue simplement comme une façon de contracter une boîte en "empilant" des sous-boîtes jusqu'à ce qu'une face soit entièrement couverte. Ce procédé est brièvement résumé dans la figure 6. Le lecteur intéressé peut se référer à [4] pour plus détails.

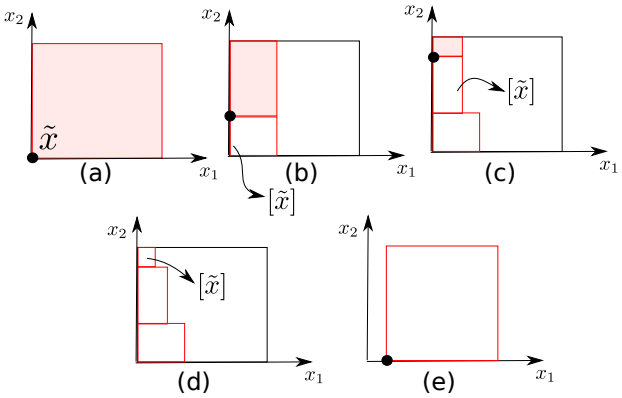


FIGURE 6 – **Boucle de sweep**. La séquence d'images illustre une contraction pour la borne inférieure de x_1. À chaque étape, le point $\tilde{x}$ étendu est le coin inférieur-gauche de la boîte grisée. La boîte intérieure $[\tilde{\mathbf{x}}]$ est en blanc. La borne inférieure de x_1 peut être réduite aussitôt que la projection des boîtes blanches sur x_2 recouvre entièrement la face $[x_2]$, ce qui est le cas à l'étape **e)**.

3.2 Contracteur Extérieur

Le contracteur extérieur est moins sophistiqué que le contracteur intérieur et agit comme un simple *test de rejet* : la boîte est entièrement éliminée ou conservée intacte.

Rejecter une boîte $[x]$ signifie prouver $[x] \not\subseteq S_1 + S_2$, c'est à dire

$$\forall x \in [x] \quad \forall p \in \mathbb{R}^d, \ \neg(c_1(p) \wedge c_2(x-p)). \tag{22}$$

Cette assertion peut être vérifiée en exécutant le même sous-solveur que pour le contracteur intérieur, à ceci près que le point $\tilde{x}$ est cette fois remplacé par la boîte courante $[x]$. Si le sous-solveur ne trouve pas de solution, l'assertion précédente est prouvée. Notons que seules les coordonnés de p sont bisectées, le sous-solveur prouve donc une assertion plus forte si la contraction par rapport à c_2 n'est pas optimale (l'assertion exacte dépend du niveau de cohérence effectuée par la contraction par rapport à c_2). Notons aussi que la précision utilisée dans le sous-solveur est dynamiquement réglée à le largeur de $[x]$ afin de maintenir une certaine uniformité dans le temps passé par le sous-solveur tout

1. Ce sous-solveur est mis en œuvre avec un branch & bound standard basé sur HC4 [3]. Puisqu'une seule solution suffit, à chaque nœud de la recherche, les inégalités sont vérifiées avec un point $\tilde{p}$ tiré aléatoirement dans le domaine courant. Si les deux sont satisfaites, la recherche est interrompue et $\tilde{p}$ est retourné. La profondeur de la recherche est également contrôlée par une précision sur la largeur du domaine, afin d'assurer une terminaison du sous-solveur en temps limité. En cas de terminaison normale, il n'a pu être trouvé de $\tilde{p}$, et donc de boîte intérieure.

au long de la recherche globale (sur x). Cette précision dynamique assure également que le contracteur extérieur est *convergent*, c'est à dire, qu'il rejette toute boîte suffisamment petite extérieure à $\mathcal{S}$.

On peut être surpris par la simplicité de ce test de rejet et s'attendre à un contracteur plus élaboré pour la région extérieure, à l'instar de celui que nous avons proposé pour la région intérieure. Mais la situation peut être interprétée dans l'autre sens. Du fait que la contrainte de chevauchement soit en deux dimensions seulement, un test de satisfiabilité intérieur serait probablement suffisant, si tant est qu'il soit assez rapide et convergent. Cependant, un tel test revient à prouver pour $[x] \subseteq S_1 + S_2$ l'assertion suivante

$$\forall x \in [x] \quad \exists p \in \mathbb{R}^d,\ (c_1(p) \wedge c_2(x-p)).$$

Or, contrairement à (22), les deux quantificateurs $\forall$ et $\exists$ sont cette fois impliqués, ce qui veut dire que le problème est beaucoup plus difficile. Donc, le contracteur intérieur peut être vu ici comme un moyen de compenser l'absence de test intérieur.

Notre argument précédent est basé seulement sur le temps d'exécution. Il est clair qu'un contracteur extérieur peut aussi conduire à une pavage plus compact, mais la taille de ce pavage est de toute façon conditionnée par la représentation de la frontière, de telle sorte qu'un gain drastique sur cet aspect ne serait de toute façon pas réellement envisageable.

4 Résultats expérimentaux

Protocole

L'algorithme que nous proposons dans cet article calcule un pavage $(\mathcal{I}, \mathcal{B}, \mathcal{O})$ de la contrainte de non-chevauchement. La difficulté de cette tache dépend principalement de trois critères :

- *occurrences des variables* : le nombre de fois où chaque variable apparaît dans les expressions des inégalités impacte directement l'efficacité du contracteur ; plus une variable apparaît, moins les contracteurs sont efficaces. C'est une limitation bien connue de l'arithmétique d'intervalles classique qui s'applique également à l'arithmétique intérieure (utilisée par le contracteur intérieur).
- *convexité* : si les objets ne sont pas convexes, la frontière de la contrainte de non-chevauchement sera moins lisse. Le pavage sera donc plus complexe (et donc plus lent à calculer), particulièrement à proximité de la frontière.
- *degrés de liberté* : c'est à dire, si nous prenons en compte la rotation ou pas. Autoriser la rotation donne lieu à un problème d'une difficulté supérieur pour plusieurs raisons. Tout d'abord, la taille du pavage étant exponentielle en le nombre de degrés de liberté, un pavage en 3D ne s'obtient pas en général dans l'échelle de temps d'un pavage 2D. De plus, les angles dans les inégalités engendrent de nombreuses multi-occurrences (voir les équations (4) et (5)) et augmentent considérablement les non-convexités par l'introduction de fonctions trigonométriques.

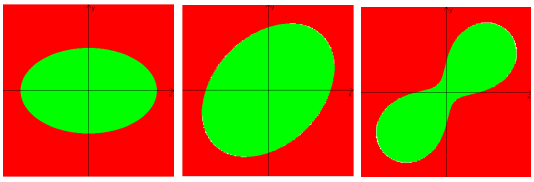


FIGURE 7 – **Objets de complexité croissante**. *De gauche à droite* : objets №1, 2 et 3.

Notre campagne de tests est basée sur ces critères. Nous avons fait deux types d'expériences. La première est avec translation seulement. Nous avons considéré trois objets de difficulté croissante. L'objet №1 est une ellipse simple :

$$\begin{array}{c}\text{Object №1 :}\\ (p_1/2)^2 + p_2^2 \leq 1.\end{array} \tag{23}$$

L'objet №2 est une ellipse "oblique" (tournée d'un angle quelconque). Les objets №1 et №2 ont évidemment la même complexité si la rotation est un degré de liberté, mais pas si nous nous limitons à translater. Cela tient au fait que l'objet tourné introduit plusieurs occurrences de p_1 et p_2 :

$$\begin{array}{c}\text{Object №2 :}\\ 1.5 \times p_1^2 + 1.5 \times p_2^2 - p_1 \times p_2 - 0.2 \leq 0.\end{array} \tag{24}$$

Finalement, le troisième objet a une forme de "cacahuète". Il cumule la multi-occurrences de variables et la non-convexité (comme on peut l'observer sur la figure 7) :

$$\begin{array}{c}\text{Object №3 :}\\ (p_1^2 + p_2^2)^2 - 2 \times (p_1 \times p_2) - 0.02 \leq 0.\end{array} \tag{25}$$

La contrainte de non-chevauchement implique deux objets : l'objet de "référence" (dont les coordonnées sont fixées à l'origine du repère) et l'objet "mobile" qui représente les inconnues. Nous avons considéré toutes les combinaisons possibles avec les trois types d'objets ci-dessus, c'est à dire, les 6 premières cases de la table 1.

Dans notre seconde série d'expériences, nous avons introduit la rotation et testé avec, d'une part, deux

Cas	Objet de ref.	Objet mobile	Rotation
1	1	1	no
2	1	2	no
3	1	3	no
4	2	2	no
5	2	3	no
6	3	3	no
7	1	1	yes
8	3	3	yes

TABLE 1 – Cas d'étude.

ellipses et, d'autre part, deux "cacahuètes" (cas 7 et 8).

Dans chaque expérience, le processus de pavage est interrompu quand la surface totale de la frontière $\mathcal{B}$ est en dessous de ε% de la surface de la boîte initiale (domaine initial pour les variables), où ε est un paramètre de précision défini par l'utilisateur. Nous avons appliqué la même politique à RSOLVER, l'outil avec lequel nous nous comparons.

Puisque que la précision est en proportion de la surface du domaine initial, il convient de noter que la qualité du pavage dépend aussi de qualité de l'enveloppe initiale. Plus le domaine initial est grand, moins le résultat final est précis. Pour cette raison, et afin donner à ε une valeur significative, nous avons initialisé dans les expériences la boîte initiale à une enveloppe assez précise de l'ensemble $\mathcal{S}$ ou $\mathcal{S}'$ (comme cela peut être vu sur les figures 8 et suivantes).

Résultats (sans rotation)

D'abord, nous comparons dans la table 2 les temps d'exécution obtenus par RSOLVER et notre algorithme pour les 6 premiers cas, avec une precision ε mise à 3.25%. Ce choix pour ε correspond à la valeur minimale avec laquelle RSOLVER produit un pavage dans les limites de temps.

Les pavages obtenus sont représentés sur la figure 8.

Nous donnons sur la figure 9 une analyse plus détaillée des temps de calcul pour les deux cas extrêmes (cas 1 et 6), où ε varie cette fois de 10% à 1%. Elles montrent des gains significatifs en performance absolue, ainsi qu'un meilleur comportement asymptotique, par rapport à RSolver.

Les résultats confirment d'abord les niveaux de complexité présumés des différent cas, puisque les instances plus "dures" exigent en effet plus de temps pour être résolues. Ils montrent aussi que notre algorithme est plus compétitif que RSOLVER. Il faut néanmoins garder à l'esprit que RSOLVER est un solveur générique, qui n'exploite pas la structure spécifique du problème que l'on traite ici. Les graphiques indiquent également que l'écart entre notre approche et RSolver,

Cas	Rsolver	Notre algorithme
1	4,07	0,37
2	51,55	2,67
3	611,85	7,90
4	132,46	5,58
5	656,11	12,00
6	771,00	26,82

TABLE 2 – Temps d'exécution (en s) pour les 6 premiers cas (la précision est à 3.25%).

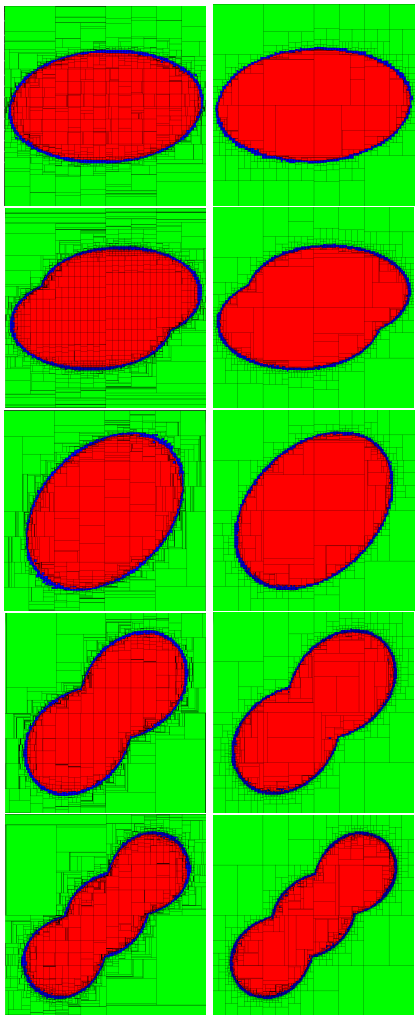


FIGURE 8 – **Pavage obtenu pour les cas 2-6**. La précision est à 3.25%. *Gauche :* avec RSolver. *Droite :* avec notre algorithme.

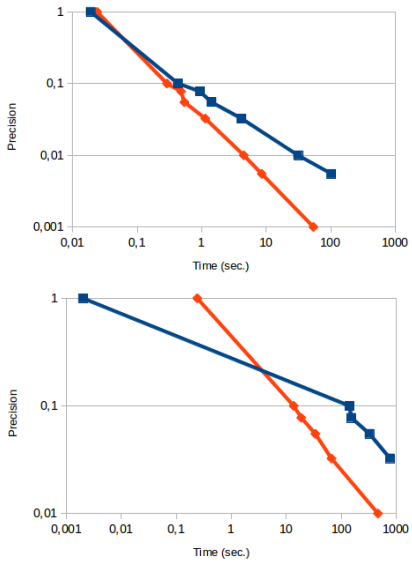


FIGURE 9 – **Temps en fonction de la précision (gauche : cas 1 ; droite : cas 6)**. Chaque courbe représente le temps de pavage (axe vertical) en fonction de la précision ε (axe horizontal). Les deux axes sont en échelle logarithmique.

pour un cas donné, augmente lorsque la valeur de la précision diminue.

Résultat (avec rotation)

Nous présentons ici seulement des résultats préliminaires avec la rotation.

Les figures 10 et 11 montrent des sections 2D des pavages 3D que nous avons obtenus dans les cas 7 et 8, avec une précision de 3.25%. Une section 2D est obtenue en fixant l'angle à une certaine valeur et en sélectionnant les boîtes dans le pavage 3D pour lesquelles l'intervalle de l'angle contient cette valeur. Les deux autres dimensions sont alors tracées.

Nous avons attribué au domaine de l'angle l'intervalle $[0, 0.7]$ pour le cas 7, et l'intervalle $[0, 0.3]$ pour le cas 8. Le seul pavage qu'il a été possible d'obtenir dans le délai imparti était avec notre algorithme, pour le cas 7. Ce pavage a été calculé en 9 minutes tandis que RSOLVER n'a pas donné de résultat après 80 minutes. Les deux algorithmes ne terminent pas au bout de 100 minutes dans le cas 8.

Quand un programme est arrêté manuellement, un résultat partiel est affiché. Un résultat partiel signifie que la surface de $\mathcal{B}$ dépasse ε% la largeur initiale.

L'objectif principal de cette expérience est de montrer que notre approche reste valide dans le cas où

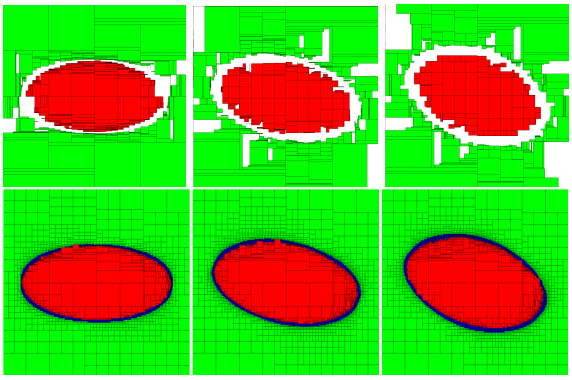


FIGURE 10 – **Section 2D d'un pavage 3D obtenu pour le cas 7.** *De gauche à droite :* les angles de rotation sont 0, 0.4 et 0.7. *Haut :* avec RSOLVER. *Bas :* avec notre algorithme.

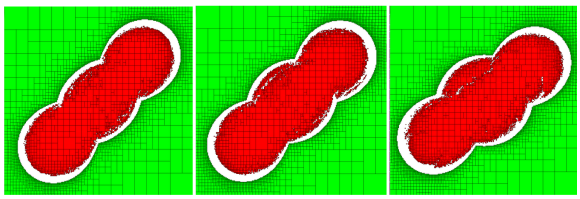


FIGURE 11 – **Section 2D d'un pavage 3D obtenu pour le cas 8.** De gauche à droite, les angles de rotation sont 0, 0.1 et 0.3. Ce pavage est calculé avec notre algorithme.

la rotation est prise en compte. La rotation implique simplement une transformation des expressions qui décrivent les objets. Elle n'exige aucun changement dans l'algorithme lui-même. Il est clair néanmoins que faire le pavage exhaustif d'un ensemble en 3D est une tâche très lourde, quel que soit l'algorithme.

5 Conclusion

Dans le cas d'objets définis par des inégalités non-linéaires, la contrainte de non-chevauchement ne peut être traitée que numériquement. Dans cet article, nous avons donné un moyen efficace pour générer une approximation garantie de cette contrainte, sous forme de pavage. Ce pavage représente explicitement la contrainte, c'est à dire, l'ensemble de toutes les positions et orientations acceptables pour un objet par rapport à un autre, déjà placé. Nos premières expériences ont montré des gains importants en efficacité par rapport à RSolver, le solveur de l'état de l'art pour les contraintes numériques quantifiées, et notamment dans

le cas où l'orientation de l'objet est prise en compte. Néanmoins, notre algorithme doit encore être amélioré dans ce cas, où les temps de calculs restent malgré tout importants. Les travaux futurs visent à améliorer encore l'efficacité de la méthode, en particulier par l'introduction d'un contracteur extérieur à la place du test de rejet extérieur actuel. Ces pavage pré-calculés seront également intégrés dans un algorithme de placement exploitant les descriptions explicites des contraintes de chevauchement.

Références

[1] I. Araya, G. Trombettoni, B. Neveu, and G. Chabert. Upper Bounding in Inner Regions for Global Optimization under Inequality Constraints. *Journal of Global Optimization*, page (to appear), 2014.

[2] N. Beldiceanu, Q. Guo, and S. Thiel. Non-Overlapping Constraints between Convex Polytopes. In *7th International Conference on Principles and Practice of Constraint Programming (CP'01)*, volume 2239 of *Lecture Notes in Computer Science*, pages 392–407. Springer-Verlag, 2001.

[3] F. Benhamou and L. Granvilliers. Continuous and interval constraints. In *Handbook of Constraint Programming*, chapter 16, pages 571–604. Elsevier, 2006.

[4] G. Chabert and N. Beldiceanu. Sweeping with Continous Domains. In D. Cohen, editor, *16th International Conference on Principles and Practice of Constraint Programming (CP'10)*, volume 6308 of *Lecture Notes in Computer Science*, pages 137–151, St Andrews, Scotland, 2010. Springer-Verlag.

[5] G. Chabert and L. Jaulin. Contractor Programming. *Artificial Intelligence*, 173(11) :1079–1100, 2009.

[6] A. Goldsztejn and L. Jaulin. Inner and Outer Approximations of Existentially Quantified Equality Constraints. In *CP*, pages 198–212. Springer, 2006.

[7] D. Ishii, A. Goldsztejn, and C. Jermann. Interval-Based Projection Method for Under-Constrained Numerical Systems. *Constraints*, 17(4) :432–460, 2012.

[8] L. Jaulin, M. Kieffer, O. Didrit, and E. Walter. *Applied Interval Analysis*. Springer, 2001.

[9] L. Jaulin and E. Walter. Set Inversion via Interval Analysis for Nonlinear Bounded-Error Estimation. *Automatica*, 29(4) :1053–1064, 1993.

[10] S. Ratschan. RSolver.

[11] S. Ratschan. Efficient Solving of Quantified Inequality Constraints over the Real Numbers. *ACM Transactions on Computational Logic*, 7(4) :723–748, 2006.

Acquisition de contraintes par requêtes de généralisation

Christian Bessiere[1] **Remi Coletta**[1] **Abderrazak Daoudi**[1,2]
Nadjib Lazaar[1] **Younes Mechqrane**[1] **El Houssine Bouyakhf**[2]

[1] CNRS, U. Montpellier, France
[2] LIMIARF/FSR, U. Mohammed V Agdal, Rabat, Maroc
{bessiere, coletta, daoudi, lazaar, mechqrane}@lirmm.fr
bouyakhf@fsr.ac.ma

Abstract

Constraint acquisition assists a non-expert user in modeling her problem as a constraint network. In existing constraint acquisition systems the user is only asked to answer very basic questions. The drawback is that when no background knowledge is provided, the user may need to answer a great number of such questions to learn all the constraints. In this paper, we introduce the concept of *generalization query* based on an aggregation of variables into types. We present a constraint generalization algorithm that can be plugged into any constraint acquisition system. We propose several strategies to make our approach more efficient in terms of number of queries. Finally we experimentally compare the recent QUACQ system to an extended version boosted by the use of our generalization functionality. The results show that the extended version dramatically improves the basic QUACQ.

Résumé

L'acquisition de contraintes assiste un novice à modéliser son problème sous forme de réseau de contraintes. Dans les systèmes d'acquisition de contraintes existants, l'utilisateur est seulement demandé à répondre à des questions très simples. L'inconvénient est que lorsqu'aucune connaissance de base n'est fournie, l'utilisateur peut avoir besoin de répondre à un grand nombre de questions pour apprendre toutes les contraintes. Dans cet article, nous introduisons le concept de *requête de généralisation* basé sur une agrégation de variables sous forme de types. Nous présentons un algorithme de généralisation de contraintes qui peut être branché dans n'importe quel système d'acquisition de contraintes. Nous proposons plusieurs stratégies pour rendre notre approche plus efficace en terme de nombre de requêtes. Finalement, nous comparons expérimentalement le récent système QUACQ à une version étendue, renforcée par l'utilisation de notre fonctionnalité de généralisation. Les résultats montrent que la version étendue améliore considérablement la version de base de QUACQ.

1 Introduction

La programmation par contraintes est utilisée pour modéliser et résoudre des problèmes combinatoires dans plusieurs domaines d'application, tels que l'allocation de ressources ou l'ordonnancement. Cependant, la modélisation sous forme de contraintes nécessite une certaine expertise en programmation par contraintes. Cela empêche l'utilisation de cette technologie par un novice ; ce qui a un effet négatif sur l'adoption de la technologie de contraintes par des non-experts.

Plusieurs techniques ont été proposées pour aider l'utilisateur dans la tâche de la modélisation. Dans [10], Freuder et Wallace ont proposé l'agent matchmaker, un processus interactif où l'utilisateur est en mesure de fournir l'une des contraintes de son problème chaque fois que le système propose une mauvaise solution. Dans [12], Lallouet et al. ont proposé un système basé sur la programmation logique inductive qui utilise les connaissances de base sur la structure du problème pour apprendre une représentation du problème classifiant correctement les exemples. Dans [4, 6], Bessiere et al. ont fait l'hypothèse que la seule chose que l'utilisateur est en mesure de fournir est des exemples de solutions et non-solutions de son problème. Sur la base de ces exemples, le système *Conacq.1* apprend un ensemble de contraintes qui classifie correctement tous les exemples donnés jusqu'ici. Ce type d'apprentissage est ap-

pelé *apprentissage passif*. Dans [3], Beldiceanu et Simonis ont proposé *ModelSeeker*, une autre approche d'apprentissage passif. Des exemples positifs sont fournis par l'utilisateur. Le système organise ces exemples sous forme d'une matrice et identifie les contraintes dans le catalogue des contraintes globales ([2]) qui sont satisfaites par des lignes ou toute autre propriété structurelle que peut capturer *ModelSeeker* de tous les exemples.

En revanche, dans un apprentissage actif comme *Conacq.2*, le système propose des exemples à l'utilisateur pour qu'il les classifient comme des solutions ou des non-solutions [7]. Ces questions sont appelées requêtes d'appartenance [1]. CONACQ présente deux défis computationnels. Le premier concerne comment le système génère une requête utile et le deuxième porte sur combien de requêtes sont nécessaires pour le système afin qu'il converge vers l'ensemble de contraintes cibles. Il a été démontré que le nombre de requêtes nécessaire pour converger vers l'ensemble des contraintes cibles peut être exponentiel [8].

QUACQ est un récent système d'apprentissage actif qui est en mesure de demander à l'utilisateur de classifier une requête *partiale* [5]. En utilisant des requêtes partielles et étant donné un exemple négatif, QUACQ est en mesure de trouver une contrainte du problème que l'utilisateur a en tête dans un nombre de requêtes logarithmiques en taille de l'exemple. Cette composante clé de QUACQ permet de converger toujours vers le réseau de contraintes cibles dans un nombre polynomial de requêtes. Cependant, malgré sa bonne borne théorique, il peut s'avérer difficile à mettre en pratique. Par exemple, QUACQ demande à l'utilisateur de classifier plus de 8000 exemples pour apprendre le modèle complet de Sudoku.

Dans cet article, nous proposons une nouvelle technique pour rendre l'acquisition de contraintes plus efficace en pratique en utilisant un typage de variables. Dans les problèmes réels, les variables représentent souvent des composantes qui peuvent être classées en différents types. Par exemple, prenons le problème d'emploi du temps scolaire, les variables peuvent représenter des enseignants, des étudiants, des salles, des cours ou des plages horaires. Ces types sont souvent connus par l'utilisateur. Pour prendre en compte les types de variables, nous introduisons la notion de *requête de généralisation* . Nous attendons de l'utilisateur d'être en mesure de décider si une contrainte apprise peut être généralisée à d'autres portées de variables du même type que ceux de la contrainte apprise. Nous introduisons l'algorithme GENACQ qui demande à l'utilisateur de classifier les requêtes de généralisation à chaque fois qu'une contrainte est apprise. Nous proposons plusieurs stratégies et heuristiques pour sélectionner la requête, meilleure candidate, à la généralisation. Nous avons bronché notre fonctionnalité de généralisation dans le système d'acquisition de contraintes QUACQ, pour obtenir l'algorithme G-QUACQ. Nous évaluons expérimentalement le bénéfice de notre technique sur plusieurs jeux de données. Les résultats montrent que G-QUACQ améliore considérablement l'algorithme de base QUACQ en terme de nombre de requêtes.

Le reste de cet article est organisé comme suit. La section 2 donne les définitions nécessaires à la compréhension de la présentation technique. La section 3 décrit l'algorithme de généralisation. Dans la section 4, plusieurs stratégies sont présentées afin de rendre notre approche plus efficace. La section 5 présente les résultats expérimentaux que nous avons obtenus lors de la comparaison de G-QUACQ à QUACQ d'une part et des différentes stratégies dans G-QUACQ d'autre part. La section 6 conclut l'article et donne quelques directions pour les futures travaux.

2 Background

Nous introduisons quelques notions utiles pour la programmation par contraintes et l'apprentissage. La connaissance commune partagée entre un apprenant qui vise à résoudre le problème et l'utilisateur qui connaît le problème est un *vocabulaire*. Ce vocabulaire est représenté par un ensemble (fini) de variables X et de domaines $D = \{D(x_1), \ldots, D(x_n)\}$ sur $\mathbb{Z}$. Une contrainte c représente une relation $rel(c)$ sur un sous-ensemble de variables $var(c) \subseteq X$ (appelé la *portée* de c) qui spécifie quelles assignations de $var(c)$ sont permises. Les problèmes combinatoires sont représentés par des *réseaux de contraintes*. Un réseau de contraintes est un ensemble C de contraintes sur le vocabulaire (X, D). Un exemple e est une assignation (partiel/complète) d'un ensemble de variables $var(e) \subseteq X$. e est rejeté par une contrainte c (i.e., $e \not\models c$) ssi $var(c) \subseteq var(e)$ et la projection $e[var(c)]$ de e sur $var(c)$ n'est pas dans $rl(c)$. Une assignation complète e de X est une solution de C ssi pour tout $c \in C$, c ne rejette pas e. On note par $sol(C)$ l'ensemble des solutions de C.

En plus du vocabulaire, l'apprenant possède un *langage* Γ de relations à partir duquel il peut construire des contraintes sur des ensembles spécifiés de variables. Un *biais de contraintes* est une collection B de contraintes construite à partir du langage Γ sur le vocabulaire (X, D).

$$B = \{c \mid (var(c) \in X) \wedge (\exists r \in \Gamma \; s.t. \; rel(c) = r \cap D^{|var(c)|})\}$$

En terme d'apprentissage automatique, un *concept* est une fonction booléenne sur $D^X = \Pi_{x_i \in X} D(x_i)$, c-à-d, une application qui attribue à chaque exemple $e \in D^X$ une valeur de $\{0, 1\}$. Le *concept cible* est le concept f_T qui renvoie 1 pour e si et seulement si e est une solution du problème que l'utilisateur a dans la tête. Dans un contexte de programmation par contraintes, le concept cible est représenté par *un réseau cible* noté C_T. Une *requête* $Ask(e)$, avec $var(e) \subseteq X$, est une question de classification posée à l'utilisateur, où e est une assignation dans

$D^{var(e)} = \Pi_{X_i \in var(e)} D(X_i)$. Un ensemble de contraintes C *accepte* une assignation e si et seulement si il n'existe aucune contrainte $c \in C$ rejetant e. La réponse à $Ask(e)$ est 'yes' *si et seulement si* C_T accepte e.

Un *type* T_i est un sous-ensemble de variables définit par l'utilisateur comme ayant une propriété commune. Une variable x est de type T_i ssi $x \in T_i$. Une portée $var = (x_1, \ldots, x_k)$ de variables appartient à une séquence $s = (T_1, \ldots, T_k)$ (noté $var \in s$) si est seulement si $x_i \in T_i$ pour tout $i \leq k$. Considérons $s = (T_1, T_2, \ldots, T_k)$ et $s' = (T'_1, T'_2, \ldots, T'_k)$ deux séquences de types. On dit que s' *couvre* s (noté $s \sqsubseteq s'$) ssi $T_i \subseteq T'_i$ pour tout $i = 1..k$. Une relation r *s'applique* sur une séquence de types s si et seulement si $(var, r) \in C_T$ pour tout $var \in s$. Une séquence de types s est *maximale* par rapport à une relation r si et seulement si r s'applique sur s et qu'il n'existe pas un s' qui couvre s sur laquelle r s'applique.

3 Algorithme GENACQ

Dans cette section, nous présentons GENACQ, un algorithme d'*acquisition généralisée*. L'idée derrière cet algorithme est, étant donné une contrainte c apprise qui s'applique sur $var(c)$, généraliser cette contrainte sur les séquences de types s qui couvrent $var(c)$ en posant les requêtes de généralisation $AskGen(s, r)$. L'utilisateur répond par 'yes' pour une requête de généralisation $AskGen(s, r)$ si et seulement si pour toute séquence var de variables couvertes par s la relation r s'applique sur var dans le réseau de contraintes cible C_T.

3.1 Description de GENACQ

L'algorithme GENACQ (voir Algorithm 1) prend comme entrée une contrainte c qui a été déjà apprise et un ensemble $NonTarget$ de contraintes qui n'appartient pas au réseau cible. Il utilise aussi la structure de données globale $NegativeQ$, qui est un ensemble de paires (s, r) pour lequel on connaît que r ne s'applique pas sur toutes les séquences de variables qui sont couvertes par s. c et $NonTarget$ peuvent venir de n'importe quel système d'acquisition de contraintes ou comme connaissances préalables. $NegativeQ$ est construite d'une manière incrémentale par chaque appel de GENACQ. GENACQ aussi utilise l'ensemble $Table$ comme structure de données locale. $Table$ va contenir toutes les séquences de types qui sont candidates pour généraliser c.

Dans la ligne 1, GENACQ initialise l'ensemble $Table$ par toutes les séquences s possibles de types qui contiennent $var(c)$. Dans la ligne 2, GENACQ initialise l'ensemble G par l'ensemble vide. G va contenir la sortie de GENACQ, c'est à dire, l'ensemble des séquences maximales de $Table$ sur lesquelles $rel(c)$ s'applique. Le compteur $\#NoAnswers$ compte le nombre de requêtes

Algorithm 1: GENACQ ($c, NonTarget$)

```
1  Table ← {s | var(c) ∈ s} \ {var(c)}
2  G ← ∅
3  #NoAnswers ← 0
4  foreach s ∈ Table do
5      if ∃(s', r) ∈ NegativeQ | rel(c) ⊆ r ∧ s' ⊑ s
       then
6          Table ← Table \ {s}
7      if
       ∃c' ∈ NonTarget | rel(c') = rel(c)∧var(c') ∈ s
8      then Table ← Table \ {s}
9  while Table ≠ ∅ ∧ #NoAnswers < cutoffNo do
10     pick s in Table
11     if AskGen(s, rel(c)) = yes then
12         G ← G ∪ {s} \ {s' ∈ G | s' ⊑ s}
13         Table ← Table \ {s' ∈ Table | s' ⊑ s}
14         #NoAnswers ← 0
       else
15         Table ← Table \ {s' ∈ Table | s ⊑ s'}
16         NegativeQ ← NegativeQ ∪ {(s, rel(c))}
17         #NoAnswers + +
18 return G ;
```

consécutives de généralisation qui sont classifiées comme négatives par l'utilisateur. Il est initialisé par zéro (ligne 3). $\#NoAnswers$ n'est pas utilisé dans la version de base de GENACQ mais il sera utilisé dans la version avec cutoffs. (Autrement dit, la version de base utilise $cutoffNo = +\infty$ dans la ligne 9).

La première boucle dans GENACQ (ligne 4) élimine de $Table$ les séquences s pour lesquelles nous connaissons déjà la réponse à la requête $AskGen(s, rel(c))$. Dans les lignes 5-6, GENACQ élimine de $Table$ toutes les séquences s telles qu'une relation r impliquée par $rel(c)$ est déjà connue qu'elle ne s'applique pas sur une séquence s' couverte par s (i.e., (s', r) appartient à $NegativeQ$). C'est sûr de supprimer ces séquences car l'absence de r sur un certaines portées dans s' implique l'absence de $rel(c)$ sur un certaines portées dans s (voir le lemme 1). Dans les lignes 7-8, GENACQ élimine de $Table$ toutes les séquences s telles que $(var, rel(c)) \notin C_T$.

Dans la boucle principale de GENACQ (ligne 9), nous sélectionnons une séquence s de $Table$ à chaque itération et nous posons la requête de généralisation à l'utilisateur (ligne 11). Si l'utilisateur dit 'yes', s est une séquence sur laquelle $rel(c)$ s'applique. Nous mettons s dans G et nous supprimons de G toutes les séquences couvertes par s, afin de garder juste la maximale (ligne 12). Nous supprimons aussi de $Table$ toutes les séquences s' couvertes par s (ligne 13) afin d'éviter de poser des questions redondantes plus tard. Si l'utilisateur dit 'no', nous sup-

primons de $Table$ toutes les séquences s' qui couvrent s (ligne 15) car nous connaissons qu'elles ne sont plus candidates pour la généralisation de $rel(c)$ et nous les mettons dans $NegativeQ$ du fait que la réponse à $(s, rel(c))$ a été 'no'. La boucle termine quand $Table$ est vide et nous renvoyons G (ligne 18).

3.2 Complétude et complexité

Nous analysons la complétude et la complexité de GENACQ en terme de nombre de requêtes de généralisation.

Lemme 1. *Si $AskGen(s, r) = no$ alors pour chaque (s', r') tel que $s \sqsubseteq s'$ et $r' \subseteq r$, nous avons $AskGen(s', r') = no$.*

Preuve. Supposons que $AskGen(s, r) = no$. Par conséquent, il existe une séquence $var \in s$ tel que $(var, r) \notin C_T$. Comme $s \sqsubseteq s'$ nous avons $var \in s'$ et nous connaissons que $(var, r) \notin C_T$. Comme $r' \subseteq r$, nous avons également $(var, r') \notin C_T$. En conséquence, $AskGen(s', r') = no$. □

Lemme 2. *Si $AskGen(s, r) = yes$ alors pour chaque s' tel que $s' \sqsubseteq s$, nous avons $AskGen(s', r) = yes$.*

Preuve. Supposons que $AskGen(s, r) = yes$. Comme $s' \sqsubseteq s$, pour toute $var \in s'$ nous avons $var \in s$ et nous connaissons que $(var, r) \in C_T$. En conséquence, $AskGen(s', r) = yes$. □

Proposition 1 (Complétude)**.** *Lorsqu'il est appelé avec la contrainte c en entrée, l'algorithme* GENACQ *renvoie toutes les séquences maximales de types couvrant $var(c)$ sur lesquelles la relation $rel(c)$ s'applique.*

Preuve. Toutes les séquences couvrant $var(c)$ sont mises dans $Table$. Une séquences dans $Table$ est soit posée pour la généralisation ou supprimée de $Table$ dans les lignes 6, 8, 13, ou 15. Nous savons par le lemme 1 qu'une séquence supprimée dans la ligne 6, 8, ou 15 conduirait nécessairement à une réponse 'no'. Nous savons du lemme 2 qu'une séquence supprimée dans la ligne 13 est subsumée et moins générale que l'autre que nous venons d'ajouter à G. □

Proposition 2. *Étant donnée une contrainte c apprise et sa $Table$ associée,* GENACQ *utilise $O(|Table|)$ requêtes de généralisation pour renvoyer toutes les séquences maximales de types couvrant $var(c)$ sur lesquelles la relation $rel(c)$ s'applique.*

Preuve. Pour chaque requête sur $s \in Table$ posée par GENACQ, la taille de $Table$ décroît strictement quelle que soit la réponse. En conséquence, le nombre total de requêtes est majoré par $|Table|$. □

3.3 Exemple illustratif

Prenons le problème de Zèbre pour illustrer notre approche de généralisation. Le problème de Zèbre de Lewis Carroll a une solution unique. Le réseau cible est formalisé en utilisant 25 variables, partitionnées en 5 types de 5 variables chacun. Les 5 types sont *color*, *nationality*, *drink*, *cigaret* et *pet*. Il y a une clique de contraintes $\neq$ sur toutes les paires de variables de même type et 14 contraintes supplémentaires figurant dans la description du problème.

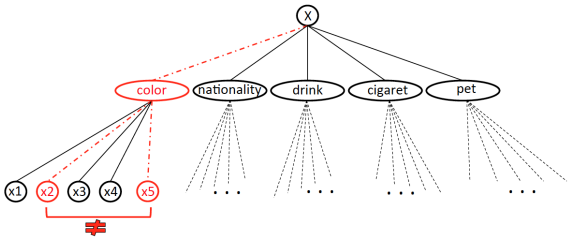


FIGURE 1 – Variables et types pour le problème de Zèbre.

La figure 1 montre les variables du problème de Zèbre et leurs types. Dans cet exemple, la contrainte $x_2 \neq x_5$ a été apprise entre les deux variables de type color x_2 et x_5. Cette contrainte est donnée comme entrée pour l'algorithme GENACQ. GENACQ calcule la $Table$ de toutes les séquences de types qui couvrent la portée (x_2, x_5). $Table = \{(x_2, color), (x_2, X), (color, x_5), (color, color), (color, X), (X, x_5), (X, color), (X, X)\}$. Supposons que nous sélectionnons $s = (X, x_5)$ à la ligne 10 de GENACQ. Selon la réponse de l'utilisateur ('no' dans ce cas), la $Table$ est réduite à $Table = \{(x_2, color), (x_2, X), (color, x_5), (color, color), (color, X)\}$. Comme prochaine itération, nous sélectionnerons $s = (color, color)$. L'utilisateur va répondre par 'yes' car il y a effectivement une clique de $\neq$ sur les variables de type $color$. Par conséquence, $(color, color)$ est ajoutée à G et $Table$ est réduite à $Table = \{(x_2, X), (color, X)\}$. Si nous sélectionnons (x_2, X), l'utilisateur répond par 'no' et nous réduisons $Table$ à l'ensemble vide et nous retournons $G = \{(color, color)\}$, ce qui signifie que la contrainte $x_2 \neq x_5$ peut être généralisée sur toutes les paires de variables dans la séquence $(color, color)$, c'est à dire, $(x_i \neq x_j) \in C_L$ pour tout $(x_i, x_j) \in (color, color)$.

3.4 Utilisation de la généralisation dans QUACQ

GENACQ est une technique générique qui peut être branchée dans n'importe quel système d'acquisition de contraintes. Dans cette section nous présentons G-QUACQ, un algorithme d'acquisition de contraintes obtenu en branchant GENACQ dans QUACQ, le système d'acquisition de contraintes présenté dans [5].

G-QUACQ est présenté dans l'algorithme 2. Comme elles apparaissent dans [5], nous ne donnons pas le code des fonctions `FindScope` et `FindC`. Mais disons quelques mots sur la façon dont elles travaillent. Étant donné les ensembles de variables S_1 et S_2, `FindScope`$(e, S_1, S_2, false)$ renvoie le sous-ensemble de S_2 qui, conjointement avec S_1, forme la portée d'une contrainte du biais B de contraintes possibles qui rejettent e. Inspiré d'une technique utilisée dans QUICKXPLAIN [11], `FindScope` requiert un nombre de requêtes logarithmiques en $|S_2|$ et linéaire en taille de la portée finale retournée. La fonction `FindC` prend comme paramètre l'exemple négatif e et la portée retournée par `FindScope`. Elle renvoie une contrainte de C_T avec la portée donnée qui rejette e. Pour chaque assignation e, $\kappa_B(e)$ décrit l'ensemble de toutes les contraintes dans B qui rejettent e.

Algorithm 2: G-QUACQ

```
1  C_L ← ∅, NonTarget ← ∅;
2  while true do
3      if sol(C_L) = ∅ then return"collapse"
4      choose e in D^X accepted by C_L and rejected by B
5      if e = nil then return "convergence on C_L"
6      if Ask(e) = yes then
7          B ← B \ κ_B(e)
8          NonTarget ← NonTarget ∪ κ_B(e)
9      else
10         c ← FindC(e, FindScope(e, ∅, X, false))
11         if c = nil then return "collapse"
12         else
13             G ← GENACQ(c, NonTarget)
14             foreach s ∈ G do
15                 C_L ← C_L ∪ {(var, rel(c)) | var ∈ s}
```

G-QUACQ a une structure très similaire à QUACQ. Il initialise l'ensemble $NonTarget$ et le réseau C_L, et va apprendre à partir de l'ensemble vide (ligne 1). Si C_L est insatisfiable (ligne 3), l'espace des réseaux possibles s'effondre parce qu'il n'existe aucun sous-ensemble du biais B donné qui est en mesure de classifier correctement les exemples déjà posés à l'utilisateur. Dans la ligne 4, QUACQ calcule une assignation complète e qui satisfait C_L mais qui viole au moins une contrainte de B. Si un tel exemple n'existe pas (ligne 5), alors toutes les contraintes dans B sont impliquées par C_L, et l'algorithme converge. Si l'algorithme ne converge pas, nous proposons l'exemple e à l'utilisateur qui va répondre par yes ou no (ligne 6). Si la réponse est yes, nous supprimons de B l'ensemble $\kappa_B(e)$ de toutes les contraintes dans B qui rejettent e (ligne 7) et nous ajoutons à l'ensemble $NonTarget$ toutes ces contraintes écartées afin de les utiliser dans GENACQ (ligne 8). Si la réponse est no, nous sommes sûr que e viole au moins une contrainte du réseau cible C_T. Nous appelons alors la fonction `FindScope` pour découvrir la portée de l'une de ces contraintes violées. `FindC` permet de sélectionner laquelle parmi les portées données est violée par e (ligne 10). Si aucune contrainte n'est retournée (ligne 11), c'est encore une condition de s'effondrer vu que nous n'avons pas trouvé dans B une contrainte qui rejette l'un des exemples négatifs. Autrement, nous savons que la contrainte c retournée par `FindC` appartient au réseau cible C_T. C'est ici que l'algorithme diffère de QUACQ vu que nous faisons appel à GENACQ pour retrouver toutes les séquences maximales de types couvrant $var(c)$ sur lesquelles $rel(c)$ s'applique. Elles sont retournées dans G (ligne 13). Ensuite, pour toute séquence de variables var appartenant à l'une de ces séquences dans G, nous ajoutons la contrainte $(var, rel(c))$ au réseau appris C_L (ligne 14).

4 Stratégies

GENACQ apprend les séquences maximales de types sur lesquelles une contrainte peut être généralisée. L'ordre dans lequel les séquences ont été sélectionné de $Table$ dans la ligne 10 de l'algorithme 1 n'est pas spécifié par l'algorithme. Comme illustré sur l'exemple suivant, les différents ordres peuvent entraîner plus ou moins rapidement aux bonnes séquences (maximales) sur lesquelles une relation r s'applique. Revenons à notre exemple sur le problème de Zèbre (Section 3.3). Dans la manière par laquelle nous développons l'exemple, nous avons besoin seulement de 3 requêtes de généralisation pour vider l'ensemble $Table$ et converger sur la séquence maximale $(color, color)$ sur laquelle $\neq$ s'applique :

1. $AskGen((X, x_5), \neq) = no$
2. $AskGen((color, color), \neq) = yes$
3. $AskGen((x_2, X), \neq) = no$

Utilisant un autre ordre, GENACQ a besoin de 8 requêtes de généralisation :

1. $AskGen((X, X), \neq) = no$
2. $AskGen((X, color), \neq) = no$
3. $AskGen((color, X), \neq) = no$
4. $AskGen((X, x_5), \neq) = no$
5. $AskGen((x_2, X), \neq) = no$
6. $AskGen((x_2, color), \neq) = yes$
7. $AskGen((color, x_5), \neq) = yes$
8. $AskGen((color, color), \neq) = yes$

Si l'on veut réduire le nombre de requêtes de généralisation, on peut se demander quelle stratégie utiliser. Dans cette section, nous présentons deux techniques. La première idée est de sélectionner les séquences dans l'ensemble $Table$ en suivant un ordre donné par une heuristique qui permet de minimiser le nombre de requêtes de généralisation. La deuxième idée est d'utiliser un cuttof sur le nombre successive de requêtes négatives auquel nous acceptons de faire face, conduisant à une stratégie de généra-

lisation non complète : la sortie de GENACQ ne sera plus garantie d'être les séquences *maximales*.

4.1 Heuristiques de sélection de requêtes

Nous proposons certaines heuristiques de sélection de requêtes basées sur les variables/contraintes impliquées dans les requêtes. Commençons par l'heuristique la plus intuitive, celle basée sur le nombre de variables impliquées dans la séquence s.

— **max_VAR :** Cette heuristique sélectionne une séquence s impliquant un nombre maximal de variables, c'est à dire, maximisant $|\bigcup_{T \in s} T|$. L'intuition derrière cette heuristique est que, si l'utilisateur répond par 'yes', un grand nombre de contraintes sera inféré.

— **min_VAR :** Cette heuristique sélectionne une séquence s impliquant un nombre minimal de variables, c'est à dire, minimisant $|\bigcup_{T \in s} T|$. L'intuition derrière cette heuristique est que, nous augmentons la chance d'avoir une réponse 'yes', et, si l'utilisateur répond par 'no', un grand nombre de séquences est supprimé de $Table$.

Nous pouvons avoir des heuristiques similaires pour le nombre de contraintes possibles impliqué dans la séquence s.

— **max_CST :** Cette heuristique sélectionne la séquence s maximisant le nombre de contraintes possibles (var, r) dans le biais, telle que var est dans s et r est la relation que nous cherchons à généraliser. L'intuition derrière cette heuristique est que, si l'utilisateur dit 'yes', la généralisation sera maximale selon le nombre de contraintes.

— **min_CST :** Cette heuristique sélectionne une séquence s minimisant le nombre de contraintes possibles (var, r) dans le biais, telle que var est dans s et r est la relation que nous cherchons à généraliser. L'intuition derrière cette heuristique est de minimiser la chance de recevoir une réponse 'no'.

Comme base de comparaison, nous définissons un sélecteur aléatoire.

— **random :** Il choisit au hasard une séquence s dans $Table$.

4.2 Utilisation des Cutoffs

L'idée ici est de quitter GENACQ avant d'avoir prouvé la maximalité des séquences retournées. Nous mettons un seuil $cutoffNo$ sur le nombre de réponses consécutives négatives afin d'éviter l'utilisation des requêtes pour vérifier des séquences peu prometteuses. L'espoir est que GENACQ renvoie les séquences quasi-maximales de types en dépit de ne pas pouvoir prouver la maximalité. Cette stratégie de cuttoff est mise en œuvre en mettant la variable $cutoffNo$ à une valeur prédéfinie. Dans les lignes 14 et 17 de GENACQ, un compteur de réponses négatives consécutives est respectivement initialisé et incrémenté selon la réponse de l'utilisateur. Dans la ligne 9, ce compteur est comparé à $cutoffNo$ afin de décider de quitter ou non.

5 Expérimentations

Nous avons fait quelques expérimentations pour évaluer l'impact de l'utilisation de notre fonctionnalité de généralisation GENACQ dans le système d'acquisition de contraintes QUACQ. Nous avons implémenté GENACQ et nous l'avons branché dans QUACQ, pour obtenir la version G-QUACQ. Nous présentons tout d'abord les jeux de données que nous avons utilisés pour nos expérimentations. Ensuite, nous présentons les résultats de plusieurs expérimentations. La première compare la performance de G-QUACQ à QUACQ de base. La deuxième présente les expérimentations évaluant les différentes stratégies que nous avons proposées (heuristiques de sélection de la requête et cuttoffs) sur G-QUACQ. La troisième évalue la performance de notre approche de généralisation quand notre connaissance sur les types de variables est incomplète.

5.1 Jeux de données

Problème de Zèbre. Comme présenté dans la section 3.3, Le problème de Zèbre de Lewis Carroll est formulé en utilisant 5 types de 5 variables chacun, avec 5 cliques de contraintes $\neq$ et 14 contraintes supplémentaires données dans la description du problème. Nous avons alimenté QUACQ et G-QUACQ par un biais B de 4450 contraintes unaires et binaires prises à partir d'un langage de 24 contraintes arithmétiques et contraintes de distance.

Sudoku. Le modèle de Sudoku est exprimé en utilisant 81 variables de domaines de taille 9, et 810 contraintes binaires $\neq$ sur les lignes, les colonnes et les carrés. Dans ce problème les types sont les 9 lignes, les 9 colonnes et les 9 carrés de 9 variables chacun. Nous avons alimenté QUACQ et G-QUACQ par un biais B de 6480 contraintes binaires prises à partir du langage $\Gamma = \{=, \neq\}$.

Carré latin. Le problème de Carré latin se compose d'une table de taille $n \times n$ dans laquelle chaque élément se produit une fois dans chaque ligne et chaque colonne. Pour ce problème, nous utilisons 25 variables de domaines de taille 5 et de 100 contraintes binaires $\neq$ sur les lignes et les colonnes. Les lignes et les colonnes sont les types de variables (10 types). Nous avons alimenté QUACQ et G-QUACQ par un biais B de contraintes basé sur le langage $\Gamma = \{=, \neq\}$.

Problème d'affectation de fréquences radio. Le problème du RLFAP consiste à fournir des canaux de communication à partir de ressources spectrales limitées [9]. Ici, nous construisons une version simplifiée du RLFAP qui consiste à distribuer toutes les fréquences disponibles sur les stations de base du réseau. Le modèle de contrainte

TABLE 1 – QUACQ vs G-QUACQ.

	QUACQ	G-QUACQ +random			
	$\#Ask$	$\#Ask$	$\#AskGen$	$\#q_P$	$\#q_N$
Zèbre	638	257	67	10	57
Sudoku	8645	260	166	42	124
Carré latin	1129	117	60	16	44
RLFAP	1653	151	37	16	21
Purdey	173	82	31	5	26

à 25 variables avec des domaines de tailles 25 et 125 contraintes binaires. Nous avons cinq stations de cinq terminaux (émetteurs/récepteurs), ce qui forment cinq types. Nous avons alimenté QUACQ et G-QUACQ par un biais B de 1800 contraintes prises à partir d'un langage de 6 contraintes arithmétiques et de distance.

Purdey. Comme le Zèbre, ce problème a une seule solution. Quatre familles ont été arrêté par le magasin général de Purdey, chacun pour acheter un article différent et payer différemment. Sous un ensemble de contraintes supplémentaires figurant dans la description, le problème est de savoir comment nous pouvons associer chaque famille a l'article qu'elle a acheté et comment elle a payé pour cela. Le réseau cible de Purdey a 12 variables avec les domaines de tailles 4 et 30 contraintes binaires. Ici, nous avons trois types de variables qui sont *family*, *bought* et *paid*, chacun d'eux contient quatre variables.

5.2 Résultats

Pour toutes nos expérimentations, nous rapportons, le nombre total $\#Ask$ de requêtes posées par QUACQ de base, le nombre total $\#AskGen$ de requêtes de généralisation et le nombre $\#q_N$ et $\#q_P$ de requêtes de généralisation successivement négatives et positives, où $\#AskGen = \#q_P + \#q_N$.

Notre première expérimentation compare QUACQ et G-QUACQ dans sa version de référence, G-QUACQ +rand, sur notre jeux de données. Le tableau 1 rapporte les résultats. Nous observons que le nombre de requêtes posées par G-QUACQ est considérablement réduit par rapport à QUACQ. Cela est particulièrement vrai sur des problèmes avec de nombreux types impliquant de nombreuses variables, tel que Sudoku ou Carré latin. G-QUACQ acquiert le Sudoku en 260 requêtes standards plus 166 requêtes de généralisation, alors que QUACQ l'acquiert en 8645 requêtes standards.

Concentrons-nous maintenant sur le comportement de nos différentes heuristiques dans G-QUACQ. Le tableau 2(haut) rapporte les résultats obtenus avec G-QUACQ en utilisant `min_VAR`, `min_CST`, `max_VAR`, et `max_CST` pour acquérir le modèle du Sudoku. (Les autres problèmes montrent des tendances similaires.) Les

TABLE 2 – G-QUACQ avec les heuristiques et la strategie cutoff sur le Sudoku.

	cutoff	$\#Ask$	$\#AskGen$	$\#q_P$	$\#q_N$
random	$+\infty$	260	166	42	124
min_VAR			90	21	69
min_CST			132	63	69
max_VAR			263	63	200
max_CST			247	21	226
min_VAR	3	260	75	21	54
	2		57	21	36
	1		**39**	21	18
min_CST	3	626	238	112	126
	2	679	231	132	99
	1	837	213	153	60

résultats montrent clairement que `max_VAR`, et `max_CST` sont de très mauvaises heuristiques. Elles sont pires que `random`. En revanche, `min_VAR` et `min_CST` surpassent significativement `random`. Elles nécessitent respectivement 90 et 132 requêtes de généralisation au lieu de 166 pour `random`. Notez qu'elles demandent tous le même nombre de requêtes standards (260) car elles ont toutes trouvé les mêmes ensembles maximaux de séquences pour chaque contrainte apprise.

En bas du tableau 2, nous comparons le comportement de nos deux meilleures heuristiques (`min_VAR` et `min_CST`) lorsqu'elles sont combinées à la stratégie de cutoff. Nous avons essayé toutes les valeurs de cutoff de 1 à 3. Une première observation est que `min_VAR` reste la meilleure quelque soit la valeur de cuttof. Fait intéressant, même avec un cutoff égale à 1, `min_VAR` nécessite le même nombre de requêtes standards que les versions de G-QUACQ sans cuttof. Cela signifie que l'utilisation de `min_VAR` comme heuristique de sélection dans $Table$, G-QUACQ est en mesure de retourner les séquences maximales même si elle est arrêtée après la première réponse négative de généralisation. Nous observons également que le nombre de requêtes de généralisation avec `min_VAR` diminue lorsque le cuttof devient plus petit (90 à 39 quand cuttof passe de $+\infty$ à 1). En regardant les deux dernières colonnes, nous voyons que c'est le nombre $\#q_N$ de réponses négatives qui diminue. La bonne performance de `min_VAR` + cutoff=1 peut donc être expliquée par le fait que `min_VAR` sélectionne en premier les requêtes qui couvrent un nombre minimal de variables ; ce qui augmente les chances d'avoir une réponse 'yes'. Finalement, nous observons que l'heuristique `min_CST` n'a pas les mêmes bonnes caractéristiques que `min_VAR`. Plus le cutoff est faible, plus le nombre de requêtes standards nécessaires devient important, pas de compensation pour l'économie de nombre de requêtes de généralisation (de 260 à 837 requêtes standards pour `min_CST` quand cutoff va de

TABLE 3 – G-QUACQ avec `random`, `min_VAR`, et cutoff=1 sur Zèbre, Carré latin, RLFAP et Purdey.

	$\#Ask$	$\#AskGen$	$\#q_P$	$\#q_N$
Zèbre				
Random	257	67	10	57
min_VAR		48	5	43
min_VAR +cutoff=1		**23**	5	18
Carré latin				
Random	117	60	16	44
min_VAR		34	10	24
min_VAR +cutoff=1		**20**	10	10
RLFAP				
Random	151	37	16	21
min_VAR		41	14	27
min_VAR +cutoff=1		**22**	14	8
Purdey				
Random	82	31	5	26
min_VAR		24	3	21
min_VAR +cutoff=1		**12**	3	9

TABLE 4 – G-QUACQ lorsque le pourcentage de types fourni augmente.

	% of types	$\#Ask$	$\#AskGen$
Zèbre	0	638	0
	20	619	12
	40	529	20
	60	417	27
	80	332	40
	100	257	48
Sudoku 9×9	0	8645	0
	33	3583	232
	66	610	60
	100	260	39
Carré latin	0	1129	0
	50	469	49
	100	117	20
Purdey	0	173	0
	33	111	8
	66	100	10
	100	82	12

$+\infty$ à 1). Cela signifie qu'avec `min_CST`, lorsque cutoff est trop petit, GENACQ ne retourne pas les séquences maximales de types où la contrainte apprise s'applique.

Dans le tableau 3, nous rapportons la performance de G-QUACQ avec `random`, avec `min_VAR` et avec `min_VAR` +cutoff=1 pour tous les autres problèmes. Nous voyons que `min_VAR` et cutoff=1 améliore significativement les performances de G-QUACQ pour tous les problèmes.

De ces expériences, nous voyons que G-QUACQ avec `min_VAR` +cutoff=1 conduit à des économies considérables en nombre de requêtes par rapport à QUACQ : 257+23 au lieu de 638 pour Zèbre, 260+39 au lieu de 8645 pour Sudoku, 117+20 au lieu de 1129 pour Carré latin, 151+22 au lieu de 1653 pour RLFAP et 82+12 au lieu de 173 pour Purdey.

Dans notre dernière expérimentation, nous montrons l'effet sur la performance de G-QUACQ lors d'un manque de connaissances sur certains types de variables. Nous avons repris nos 5 jeux de données dans lesquels nous avons fait varier le pourcentage de types connus par l'algorithme. Cela simule une situation dans laquelle l'utilisateur ne sait pas que certaines variables sont du même type. Par exemple, dans le Sudoku, l'utilisateur n'a pas pu remarqué que les variables sont regroupées en colonnes. La Figure 2 présente le nombre de requêtes standards et requêtes de généralisation posées par G-QUACQ avec `min_VAR` +cutoff=1 pour apprendre le modèle de RLFAP lorsqu'il est alimenté par une connaissance de plus en plus précise des types. Nous observons que dès qu'un petit pourcentage de types est connu (20%), G-QUACQ réduit considérablement le nombre de requêtes. Le tableau 4 donne la même information pour tous les autres problèmes.

6 Conclusion

Nous avons proposé une nouvelle technique pour rendre l'acquisition de contraintes plus efficace en utilisant des informations sur les types de composantes des variables représentants le problème. Nous avons introduit les requêtes de généralisation, un nouveau type de requête qui demande à l'utilisateur de généraliser une contrainte à d'autres portées de variables du même type où cette contrainte éventuellement s'applique. Notre nouvelle technique, GENACQ, peut être appelée à généraliser chaque

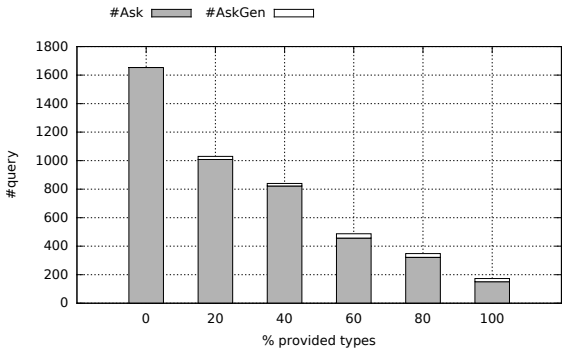


FIGURE 2 – G-QUACQ sur RLFAP lorsque le pourcentage de types fourni augmente.

nouvelle contrainte apprise par un système d'acquisition de contraintes. Nous avons proposé plusieurs heuristiques et stratégies pour sélectionner la requête, meilleure candidate à la généralisation. Nous avons branché GENACQ dans le système d'acquisition de contraintes QUACQ, pour obtenir l'algorithme G-QUACQ. Nous avons évalué expérimentalement le bénéfice de notre approche sur plusieurs jeux de données, avec et sans connaissance complète sur les types de variables. Les résultats montrent que G-QUACQ améliore considérablement l'algorithme de base QUACQ en terme de nombre de requêtes.

Références

[1] Dana Angluin. Queries and concept learning. *Machine Learning*, 2(4) :319–342, 1987.

[2] Nicolas Beldiceanu, Mats Carlsson, Sophie Demassey, and Thierry Petit. Global constraint catalogue : Past, present and future. *Constraints*, 12(1) :21–62, 2007.

[3] Nicolas Beldiceanu and Helmut Simonis. A model seeker : Extracting global constraint models from positive examples. In Michela Milano, editor, *Principles and Practice of Constraint Programming - 18th International Conference, CP 2012, Québec City, QC, Canada, October 8-12, 2012. Proceedings*, volume 7514 of *Lecture Notes in Computer Science*, pages 141–157. Springer, 2012.

[4] Christian Bessiere, Remi Coletta, Eugene C. Freuder, and Barry O'Sullivan. Leveraging the learning power of examples in automated constraint acquisition. In Mark Wallace, editor, *Principles and Practice of Constraint Programming - CP 2004, 10th International Conference, CP 2004, Toronto, Canada, September 27 - October 1, 2004, Proceedings*, volume 3258 of *Lecture Notes in Computer Science*, pages 123–137. Springer, 2004.

[5] Christian Bessiere, Remi Coletta, Emmanuel Hebrard, George Katsirelos, Nadjib Lazaar, Nina Narodytska, Claude-Guy Quimper, and Toby Walsh. Constraint acquisition via partial queries. In Francesca Rossi, editor, *IJCAI 2013, Proceedings of the 23rd International Joint Conference on Artificial Intelligence, Beijing, China, August 3-9, 2013*. IJCAI/AAAI, 2013.

[6] Christian Bessiere, Remi Coletta, Frédéric Koriche, and Barry O'Sullivan. A sat-based version space algorithm for acquiring constraint satisfaction problems. In João Gama, Rui Camacho, Pavel Brazdil, Alípio Jorge, and Luís Torgo, editors, *Machine Learning : ECML 2005, 16th European Conference on Machine Learning, Porto, Portugal, October 3-7, 2005, Proceedings*, volume 3720 of *Lecture Notes in Computer Science*, pages 23–34. Springer, 2005.

[7] Christian Bessiere, Remi Coletta, Barry O'Sullivan, and Mathias Paulin. Query-driven constraint acquisition. In Manuela M. Veloso, editor, *IJCAI 2007, Proceedings of the 20th International Joint Conference on Artificial Intelligence, Hyderabad, India, January 6-12, 2007*, pages 50–55, 2007.

[8] Christian Bessiere and Frédéric Koriche. Non learnability of constraint networks with membership queries. Technical report, Coconut, Montpellier, France, February, 2012.

[9] Bertrand Cabon, Simon de Givry, Lionel Lobjois, Thomas Schiex, and Joost P. Warners. Radio link frequency assignment. *Constraints*, 4(1) :79–89, 1999.

[10] Eugene C. Freuder and Richard J. Wallace. Suggestion strategies for constraint-based matchmaker agents. *International Journal on Artificial Intelligence Tools*, 11(1) :3–18, 2002.

[11] Ulrich Junker. Quickxplain : Preferred explanations and relaxations for over-constrained problems. In Deborah L. McGuinness and George Ferguson, editors, *Proceedings of the Nineteenth National Conference on Artificial Intelligence, Sixteenth Conference on Innovative Applications of Artificial Intelligence, July 25-29, 2004, San Jose, California, USA*, pages 167–172. AAAI Press / The MIT Press, 2004.

[12] Arnaud Lallouet, Matthieu Lopez, Lionel Martin, and Christel Vrain. On learning constraint problems. In *22nd IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2010, Arras, France, 27-29 October 2010 - Volume 1*, pages 45–52. IEEE Computer Society, 2010.

Une Recherche Tabou pour le Problème d'Intégration de Services Réseau En et Hors Ligne

Quang Dung Pham[1] **Florence Massen**[2] **Yves Deville**[2] **Olivier Bonaventure**[2]

[1] School of Information & Communication Technology, Hanoi University of Science and Technology
[2] ICTEAM, Université catholique de Louvain, Belgium

dungpq@soict.hut.edu.vn
{Florence.Massen, Yves.Deville, Olivier.Bonaventure}@uclouvain.be

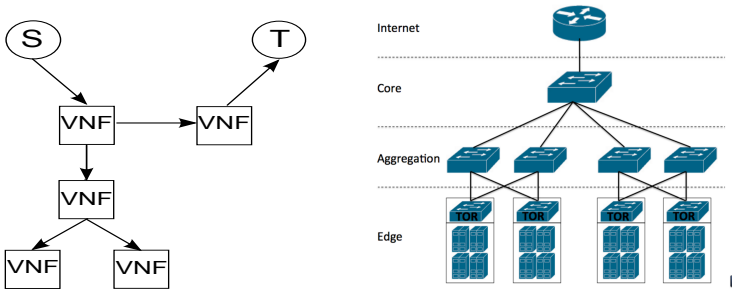
Résumé

Le problème d'intégration de services réseau se pose dans le cadre d'un nouveau paradigme dans l'industrie des télécommunications. Des fonctions réseau, typiquement assurées par des dispositifs hardware, sont virtualisées et regroupées dans des services réseau. Ces services réseau sont ensuite exécutés dans des centres de traitement de données standards. Le problème que nous abordons est l'intégration d'un ensemble de services réseau dans un centre de traitement de données de manière à satisfaire des contraintes de ressources et de performance. Dans cet article, nous considérons à la fois la variante hors ligne et en ligne de ce problème. Nous présentons différentes heuristiques pour générer des solutions initiales et une recherche tabou pour trouver une solution satisfaisant toutes les contraintes du problème. La viabilité de notre approche est illustrée dans plusieurs expériences.

1 Introduction

Les réseaux TCP/IP traditionnels ont été construits en mettant ensemble un grand nombre de routeurs, de switches (commutateurs) et différents types de liens. Au cours de la dernière décennie, l'organisation de ces réseaux a considérablement changé. Les réseaux d'entreprise d'aujourd'hui contiennent différents types d'équipement dont les fonctionnalités diffèrent de celles des switches et des routeurs traditionnels. Une enquête récente sur les réseaux d'entreprise ([16]) a révélé que ces réseaux contiennent plus de middleboxes (boîtiers de médiation) que de routeurs et de switches. Ces middelboxes implémentent des fonctions spécifiques (traduction d'adresses NAT, pare-feu, sécurité,...) qui nécessitent chacune du matériel hardware dédié. Les fournisseurs d'accès à Internet (FAI) et les réseaux mobiles 3G/4G se retrouvent face à une révolution semblable. Le déploiement de matériel dédié pour chaque nouveau service implique d'importants coûts d'installation et de fonctionnement.
De nombreux opérateurs réseau sont à la recherche de solutions pour réduire ces coûts. Du côté serveur, la virtualisation a permis le développement de services Internet échelonnables ([9]) sur une infrastructure composée de serveurs x86 et de switches (pour une vue d'ensemble de l'état-de-l'art de la virtualisation de réseaux cf. [2]).
Il a été montré ([7, 3]) qu'une infrastructure similaire peut fournir des services réseau (SR) tels que traduction d'adresse, routeur, pare-feu etc. Les opérateurs réseau ont adopté cette approche et au cours des dernières années, plusieurs d'entre eux ont travaillé sur la spécification de la virtualisation des fonctions réseau (VFR, network functions virtualization) dans ETSI, cf. [4]. On s'attend à ce que la VFR se dégage comme approche principale pour le déploiement de nouveaux services dans les grands réseaux des FAIs et 3G/4G. Avec VFR, les fonctions réseau sont implémentées avec des logiciels exécutés sur des serveurs x86 virtualisés et interconnectés par des switches à haute performance. Pour obtenir de bonnes performances, ces fonctions réseau virtualisées (VNFs) sont typiquement assez petites. Un service réseau (SR) se compose de séquences de VNFs (décrites par le graphe de transmission) qui sont appliquées sur le flux (de paquets). Par exemple, un émetteur dans un réseau privé virtuel pourrait être composé d'une fonction de pare-feu pour détecter tout problème dans le flux de paquets, suivie d'une fonction de compression pour réduire la taille du paquet et une fonction de chiffrement.

En pratique, les serveurs x86 et les switches feront partie d'un data center (centre de traitement de données) et le déploiement d'un nouveau SR nécessite d'une part la sélection des switches et des serveurs qui accueilleront les



(a) Un service réseau pour la détection d'intrusions avec 1 source (S), 1 puit (T) et 5 VNFs.

(b) Un data centeravec 4 racks (Edge), 4 switches intermédiaires (Aggregation) et 1 I/O-switch (Internet)

FIGURE 1 – Illustration d'un service réseau (SR) et d'un data center

différentes composantes du SR et d'autre part la sélection de liens dans le data center qui seront utilisés pour gérer le flux entre les composantes du SR. Ce problème d'intégration de SRs dans des data centers (PISR, Problème d'Intégration de Services Réseau) est au cœur du présent document.

Chaque SR est représenté par le graphe de transmission, décrivant le flux entre les différentes VNFs du service. Ce graphe est relié à des noeuds de source et de puit représentant les extrémités du SR (par exemple interface vers l'Internet ou un réseau). Un exemple d'un graphe de transmission d'un SR avec source et puit est représenté à la Figure 1a. Ce service assure la détection d'intrusions. La source (S), représente le point d'entrée d'un flux. Ce flux s'écoule ensuite à travers différentes VNFs qui analysent les paquets, les écartent ou les transmettent à la prochaine VNF. Enfin, les paquets restants sont acheminés vers le puit (T), où ils s'écoulent de nouveau vers le monde extérieur. Notez que toutes les VNFs ne sont pas connectées au puit, comme certaines d'entre elles sont utilisées uniquement à des fins d'enregistrement. Les SRs sont composés d'un nombre variable de sources, puits et VNFs, reliés entre eux par des arcs représentant le flux. L'exécution d'un SR dans un data center nécessite son intégration dans ce data center. Un petit data center est illustré à la Figure 1b. Le niveau le plus bas (Edge) est composé de 16 serveurs répartis sur 4 racks dont chacun est lié à un top-of-rack switch (ToR-switch). Au plus haut niveau (Internet) un I/O-switch relie le data center avec l'Internet. Les data centers sont composés d'un nombre variable de serveurs, racks, ToR-switches, switches intermédiaires, switches centraux et d'I/O-switches.

Il est clair qu'aussi bien les SRs que les data centers peuvent être représentés sous forme de graphes (pour simplifier la compréhension nous utiliserons SR et data center aussi bien pour désigner les concepts que pour désigner les graphes les décrivant). L'intégration d'un ensemble de SRs dans le data center nécessite alors un mappage plusieurs-à-un des noeuds des graphes de tous les SRs à des noeuds du graphe du data center et un mappage plusieurs-à-un des arcs des graphes de tous les SRs vers des chemins dirigés sur le graphe du data center. Pour être faisable, ces mappages doivent satisfaire un ensemble de contraintes de ressources et de performance.
Dans cet article, nous considérons à la fois la variante *hors ligne* et la variante *en ligne* du PISR. Dans la variante hors ligne l'ensemble des SRs à être intégrés dans le data center est connu à l'avance. Dans la variante en ligne, les demandes d'intégration de SRs apparaissent au fil du temps, et pour chaque demande le système doit décider si le SR peut être intégré faisablement dans le data center, compte tenu des SR déjà intégrés.
La contribution de cet article est double. D'abord nous présentons un problème pertinent, rencontré dans le cadre d'un changement de paradigme dans la branche des télécommunications. Le problème est central pour la mise en oeuvre de la virtualisation des fonctions réseau. Puis, nous présentons une recherche tabou et plusieurs procédures de génération de solution initiale pour les variantes en et hors ligne du problème. Nous montrons que pour la variante en ligne notre méthode est capable de gérer un grand nombre de SRs sur un grand data center, tout en accordant seulement un court laps de temps d'exécution à l'intégration de chaque nouveau SR entrant.

Le reste de cet article est organisé comme suit : d'abord, une définition formelle du problème considéré est donnée dans la section 2. La section 3 présente notre méthodologie pour résoudre la variante hors ligne. La section 4 explique comment cette méthodologie peut être adaptée pour traiter le problème en ligne. Ensuite, nous présentons des résultats expérimentaux pour les variantes hors ligne et en ligne dans la section 5. Enfin un aperçu des travaux apparentés pertinents est donné dans la section 6.

2 Description du problème

Le PISR peut être considéré comme un problème de mapping entre noeuds et de mapping entre arcs et chemins dirigés. S'agissant d'un problème de satisfaction de

contraintes, l'objectif est de trouver une solution faisable. Dans cette section nous décrivons les différentes composantes du problème sous forme de graphes, expliquons ce qui constitue une solution au PISR et détaillons les contraintes qu'une solution faisable doit respecter.

2.1 Composantes du problème

Le *data center* peut être représenté par un graphe orienté (avec boucles) $\mathscr{D} = (F,A)$. Les noeuds de F sont répartis en noeuds I/O-switch $F_{\texttt{I/O}}$, autres noeuds switch $F_{\texttt{SW}}$ et les noeuds serveur $F_{\texttt{SERV}}$. Nous avons donc $F = F_{\texttt{I/O}} \cup F_{\texttt{SW}} \cup F_{\texttt{SERV}}$. Avec chaque noeud serveur $f_s \in F_{\texttt{SERV}}$ est associé un certain nombre de coeurs CPU et une certaine capacité de mémoire, C_S et M_S. Tout arc $a_i \in A$ a deux attributs : sa bande passante b_i et sa latence D_i. Chaque serveur du data center dispose d'une bande passante interne qui est modélisée à l'aide d'une boucle. La bande passante interne B_s d'un serveur $f_s \in F_{\texttt{SERV}}$ correspond à la bande passante de l'arc $a_s \in A$, représentant une boucle sortant de et entrant au noeud f_s.
Chaque *SR* est représenté par un graphe acyclique orienté $\mathscr{S} = (G,H,T)$ avec G l'ensemble des noeuds, H l'ensemble des arcs et T un ensemble de chemins dirigés. Les noeuds de G sont répartis en source $G_{\texttt{SOURCE}}$, puit $G_{\texttt{SINK}}$ et noeuds VNF $G_{\texttt{VNF}}$. Nous avons donc $G = G_{\texttt{SOURCE}} \cup G_{\texttt{VNF}} \cup G_{\texttt{SINK}}$. Si deux noeuds dans ce graphe sont reliés par un arc, cela signifie qu'il existe un flux de paquets d'un noeud à l'autre. Le graphe ne contient pas d'arcs entrant dans des noeuds de $G_{\texttt{SOURCE}}$ ni d'arcs sortant des noeuds de $G_{\texttt{SINK}}$. Les chemins dirigés dans T sont des chemins entre les noeuds de source et de puit sur lesquels une latence maximale ne doit pas être dépassée. Cette latence maximale z_u est donnée dans la définition du problème pour chaque chemin $t_u \in T$.
Avec chaque noeud $g_p \in G_{\texttt{VNF}}$ représentant une VNF, sont associés une demande en coeurs CPU et une demande en mémoire c_p et m_p. Puis un noeud I/O-switch $U_{so} \in F_{\texttt{I/O}}$ du data center est associé avec chaque noeud de source $g_{so} \in G_{\texttt{SOURCE}}$. U_{so} est le seul noeud I/O-switch qui peut être alloué à g_{so}. De même, un sous-ensemble de noeuds $V_{si} \subseteq F_{\texttt{I/O}}$ est donné pour chaque noeud puit $g_{si} \in G_{\texttt{SINK}}$, et g_{si} ne peut être placé que sur un des I/O-switches de l'ensemble V_{si}. U_{so} et V_{si} dépendent du SR et sont donnés dans la définition du problème. Enfin, chaque arc $h_i \in H$ du SR est associée une demande en bande passante b_i. Elle spécifie la bande passante consommée sur chaque arc du data center implémentant le flux représenté par l'arc h_i.
On note l'*ensemble des graphes de tous les SRs*, représentant les SRs qui doivent être intégrés dans le data center, par $\Xi = \{\mathscr{S}^1, \ldots, \mathscr{S}^n\}$ avec $\mathscr{S}^i = (G^i, H^i, T^i), 1 \leq i \leq n$.

2.2 Solution au problème

Une solution au problème considéré est composée de deux parties : d'une part un mapping plusieurs-à-un de tous les noeuds de tous les SRs aux noeuds du data center et d'autre part un mapping plusieurs-à-un de tous les arcs de tous les SRs à des chemins dirigés dans le data center. D'abord, un mapping entre les noeuds des SRs et les noeuds du data center peut être défini comme une fonction $\texttt{s} : G \rightarrow F$ qui assigne un noeud du data center $\texttt{s}(g_i)$ à chaque noeud de SR $g_i \in G$ tel que :

- $g_i \in G_{\text{SOURCE}} \Rightarrow \texttt{s}(g_i) = U_i$
- $g_i \in G_{\text{VNF}} \Rightarrow \texttt{s}(g_i) \in F_{\texttt{SERV}}$
- $g_i \in G_{\text{SINK}} \Rightarrow \texttt{s}(g_i) \in V_i$

Notez qu'il est possible d'assigner plusieurs noeuds de (plusieurs) SR à un seul noeud du data center (e.g. assignation de plusieurs VNFs à un seul serveur).

Puis un mapping des arcs des SRs sur des chemins dirigés dans le data center doit être trouvé. Un chemin dirigé est défini comme une séquence d'arcs, de sorte que, pour toute paire d'arcs consécutifs dans la séquence, la destination du premier arc est l'origine du deuxième arc. Ces chemins dirigés dans le data center seront utilisés pour implémenter le flux entre les composantes d'un SR.
Étant donné que la fonction $\texttt{s}$ fait déjà le mapping des noeuds des SRs sur les noeuds du data center, nous connaissons pour chaque arc d'un SR, le noeud d'origine et le noeud de destination du chemin dirigé correspondant dans le graphe du data center. Ainsi, tout ce qu'il faut en plus est une fonction, qui, recevant une paire de noeuds du data center, fournit un chemin dirigé entre ces deux noeuds. Soit $\mathscr{P}$ l'ensemble de tous les chemins dirigés dans le graphe du data center $\mathscr{D}$ et soit l'ensemble de noeuds $F_p = (F_{\texttt{SERV}} \times F_{\texttt{SERV}}) \cup (F_{\texttt{I/O}} \times F_{\texttt{SERV}}) \cup (F_{\texttt{SERV}} \times F_{\texttt{I/O}})$.
Avec $\texttt{s}(g_i) = f_i$ et $\texttt{s}(g_j) = f_j$ et $f_i, f_j \in F_p$, la fonction $\texttt{path} : F_p \rightarrow \mathscr{P}$ assigne alors à tout arc $(g_i, g_j) \in H$ de tout SR $\mathscr{S} = (G,H,T)$ ($\mathscr{S} \in \Xi$) un chemin dirigé $\texttt{path}(f_i, f_j) \in \mathscr{P}$.
Une solution faisable doit respecter un ensemble de contraintes spécifiées dans la prochaine section.

2.3 Contraintes du problème

Une solution est dite faisable si elle respecte un ensemble de contraintes données. Afin de mieux décrire ces contraintes nous introduisons les notations suivantes : Soit *sol* une solution représentée par $\texttt{s}_{sol}$ et $\texttt{path}_{sol}$:

- $G^{\Xi} = \bigcup_{i=1}^{n} G^i$
- $H^{\Xi} = \bigcup_{i=1}^{n} H^i$
- $T^{\Xi} = \bigcup_{i=1}^{n} T^i$
- $\texttt{UTIL}(a \in A) = \{(g,g') \in H^{\Xi} : a \in \texttt{path}_{sol}(\texttt{s}_{sol}(g), \texttt{s}_{sol}(g'))\}$
- $\texttt{LAT}((g,g') \in H^{\Xi}) = \sum_{a_i \in \texttt{path}_{sol}(\texttt{s}_{sol}(g), \texttt{s}_{sol}(g'))} D_i$

Les contraintes à respecter sont des contraintes de type ressource sur les CPUs, la mémoire (éq. 1 et 2) et sur la bande passante (éq. 3) ainsi que des contraintes de perfomance

sur la latence maximale (éq. 4) :

$$\sum_{g_i \in G^{\Xi} : \mathtt{s}(g_i)=f_j} c_i \leq C_j \qquad \forall f_j \in F_{\mathtt{SERV}} \qquad (1)$$

$$\sum_{g_i \in G^{\Xi} : \mathtt{s}(g_i)=f_j} m_i \leq M_j \qquad \forall f_j \in F_{\mathtt{SERV}} \qquad (2)$$

La contr. 1 interdit que la somme des demandes de coeurs CPU de tous les noeuds VNF (sur tous les SRs dans l'ensemble Ξ) assignés à un noeud serveur f_j excède le montant de coeurs CPU de ce serveur. La contr. 2 est analogue en termes de mémoire demandée.

$$\sum_{h_i \in \mathtt{UTIL}(a_j)} b_i \leq B_j \qquad \forall a_j \in A \qquad (3)$$

La contr. 3 interdit la bande passante totale consommée sur un arc du graphe du data center d'excéder sa bande passante. La bande passante totale du flux dépend aussi bien du mapping entre noeuds que du mapping entre arcs et chemins dirigés.

$$\sum_{h \in t_u} \mathtt{LAT}(h) \leq z_u \qquad \forall t_u \in T^{\Xi} \qquad (4)$$

Un chemin dirigé $t \in T$ dans le graphe d'un SR $\mathscr{S} = (G,H,T)$ est une séquence d'arcs dans H. Pour tout arc dans H, une solution choisit un chemin dirigé dans le graphe du data center (fonction `path`) pour implémenter le flux de données représenté par l'arc dans H. Donc, un chemin dirigé dans $\mathscr{S}$ correspond également à un chemin dirigé dans le graphe du data center. Chaque arc a_q dans le graphe du data center $\mathscr{D}$ possède une certaine latence D_q. La contr. 4 interdit, pour chaque chemin dirigé $t_u \in T^{\Xi}$ sur un SR, que la latence cumulée sur les arcs du data center qui implémentent t_u excède la latence maximale z_u du chemin dirigé t_u sur le SR.

3 Méthodologie problème hors ligne

Dans cette section, nous présentons un algorithme de recherche tabou pour la variante hors ligne du PISR. Dans cette variante l'ensemble Ξ des SRs à intégrer dans le data center est connu dès le début. PISR est un problème de satisfaction de contraintes. Le but est donc de trouver une solution faisable, correspondant à un mapping de tous les noeuds de tous les SRs aux nœuds du data center et de tous arcs de tous les SRs vers des chemins dirigés sur le data center. L'ensemble des chemins dirigés sur le data center est déterminé dans une étape de précalcul où un chemin dirigé entre chaque paire de noeuds intéressants du data center est décidé. Compte tenu de cet ensemble de chemins précalculés, un mapping des noeuds des SRs vers des noeuds du data center correspond en même temps à un mapping des arcs des SRs vers des chemins dirigés sur le data center.
Afin de trouver une solution faisable, il suffit alors de trouver un mapping entre les noeuds des SRs et les nœuds du data center, de façon à ce que les contraintes (1) à (4) soient respectées compte tenu de l'ensemble de chemins dirigés précalculés sur le data center. Nous proposons une recherche tabou, qui, à partir d'une solution initiale infaisable, vise à minimiser le nombre de violations de contraintes.

Par la suite nous présentons d'abord l'algorithme utilisé pour précalculer les chemins sur le data center. Ensuite, une heuristique gloutonne pour générer une solution initiale (éventuellement infaisable) est présentée. Enfin, la recherche tabou est décrite en détail. Avant tout nous introduisons quelques notations et concepts utilisés dans le reste de cet article.

3.1 Notations et mesures de violation

Nous utilisons *sol* comme notation courte pour dénoter une solution au problème. La solution *sol* correspond à une fonction de mapping $\mathtt{s}_{sol}$ et une fonction $\mathtt{path}_{sol}$ qui assigne des chemins dirigés (précalculés) à des arcs des SRs. Notez que, étant donné l'ensemble de chemins précalculés, la fonction $\mathtt{path}_{sol}$ peut être dérivée directement de la fonction $\mathtt{s}_{sol}$, et donc la fonction $\mathtt{s}_{sol}$ et l'ensemble de chemins précalculés sont suffisants pour complétement décrire une solution *sol*. Le nombre de violations des contraintes dans une solution *sol* est donné par $v(sol) = v_{cpu}(sol) + v_{mem}(sol) + v_{bp}(sol) + v_{lat}(sol)$ avec :

- $v_{cpu}(sol) = \sum_{f_j \in F_{\mathtt{SERV}}} max(0, \sum_{g_i \in G^{\Xi}\, \mathtt{s}_{sol}(g_i)=f_j} c_i - C_j)$
- $v_{mem}(sol) = \sum_{f_j \in F_{\mathtt{SERV}}} max(0, \sum_{g_i \in G^{\Xi}\, \mathtt{s}_{sol}(g_i)=m_j} c_i - M_j)$
- $v_{bp}(sol) = \sum_{a_j \in A} max(0, \sum_{h_i \in \mathtt{UTIL}(e_j)} b_i - B_j)$
- $v_{lat}(sol) = \sum_{t_u \in T} max(0, \sum_{h \in t_u} \mathtt{LAT}(h) - z_u)$

3.2 Précalcul de chemins dirigés

Au stade du précalcul aucun mapping entre les noeuds des SRs et les noeuds du data center est disponible, et nous ne savons donc pas entre quelle paire de noeuds du data center un chemin dirigé doit être trouvé. C'est pour cette raison qu'un chemin dirigé est calculé entre chaque paire de noeuds de serveur, et entre chaque paire noeud serveur et noeud I/O-switch. Ces chemins sont déterminés de façon à ce que le nombre de chemins qui passent par chaque arc du data center soit équilibré. Cela signifie que nous essayons d'éviter des situations dans lesquelles un arc du data center est partagé par de nombreux chemins tandis qu'un autre arc est utilisé par seulement peu de chemins.
Le pseudo-code pour le calcul des chemins est donné dans l'algorithme 1. La première étape est le calcul de F_p, l'ensemble de paires de noeuds du data center entre lesquels un chemin doit être trouvé (ligne 1). Puis un coût de 1 est associé à chaque arc dans le graphe $\mathscr{D}$ (ligne 2). Ensuite, des paires de noeuds sont choisies au hasard dans F_p (ligne 4). Pour chaque paire de noeuds, le plus court chemin dans le

graphe $\mathscr{D}$, tenant compte du coût $c(a)$ de chaque arc $a \in A$, est calculé (ligne 5). Le coût des arcs utilisés par ce chemin est ensuite incrémenté (ligne 7), ceci dans l'optique de dissuader de l'utilisation de ces arcs dans le calcul du plus court chemin pour une différente paire de noeuds. Cette procédure est répétée pour chaque paire de noeuds dans l'ensemble F_p.

Algo. 1: Calcul de chemins dirigés dans le data center

Output : P l'ensemble de chemins dirigés entre noeuds dans F_p

1 $F_p = (F_{\texttt{SERV}} \times F_{\texttt{SERV}}) \cup (F_{\texttt{I/O}} \times F_{\texttt{SERV}}) \cup (F_{\texttt{SERV}} \times F_{\texttt{I/O}})$;
2 **foreach** $a \in A$ **do** $c(a) \leftarrow 1$ **while** $F_p \neq \emptyset$ **do**
3 $\quad \langle v_1, v_2 \rangle \leftarrow$ Pop aléatoire de F_p;
4 $\quad p(v_1, v_2) \leftarrow$ chemin le plus court de v_1 à v_2;
5 $\quad P \leftarrow P \cup p(v_1, v_2)$;
6 $\quad$ **foreach** $a \in p(v_1, v_2)$ **do** $c(a) \leftarrow c(a) + 1$
7 **end**
8 **return** P;

3.3 Solution initiale avec heuristique round-robin

Cette section décrit un algorithme round-robin glouton pour établir une solution initiale. Le but de l'algorithme est, étant donné un ensemble précalculé de chemins P sur le data center, de trouver un mapping entre les noeuds des SRs dans Ξ et les noeuds du data center, de telle sorte que le nombre de violations des contraintes soit aussi petit que possible. Afin de faciliter le respect des contraintes de latence, l'algorithme tente le plus possible de placer les VNFs d'un même SR sur un même serveur, ou sur un ensemble de serveurs dans un même rack (tous connectés à un même ToR-switch).

L'algorithme commence par placer chaque noeud source $g_s \in G^i_{\texttt{SOURCE}}$ de chaque SR $\mathscr{S}^i$ sur l'I/O-switch U^i_s et chaque noeud puit $g_t \in G^i_{\texttt{SINK}}$ sur un I/O-switch choisi au hasard dans V^i_t. Ensuite, chaque SR $\mathscr{S}^i \in \Xi$ et chacune des ses VNF $g \in G_{\texttt{VNF}}$ sont considérés individuellement. Supposons que les serveurs du data center soient numérotées de $1, \ldots, |F_{\texttt{SERV}}|$, et que les serveurs dans un même rack soient numérotés consécutivement. L'algorithme maintient un pointeur p (initialisé au serveur 1) sur le serveur courant. L'approche gloutonne tente d'abord de placer le VNF courant g sur le serveur courant p. Si cela introduit de nouvelles violations de contraintes, l'algorithme essaie le prochain serveur $p+1$. De cette façon, tous les serveurs $p, \ldots, |F_{\texttt{SERV}}|, 1, \ldots, p-1$ sont considérés jusqu'à ce qu'un serveur soit trouvé sur lequel la VNF g puisse être placé ; p est alors mis à jour pour pointer vers ce serveur. Si un tel serveur ne peut pas être trouvé, nous choisissons le serveur qui crée le moins de violations quand g lui est attribué et le pointeur p est déplacé vers ce serveur. Cette procédure est répétée pour chaque VNF de chaque SR. La solution résultante est éventuellement infaisable. Dans ce cas, la recherche tabou est utilisée pour la rendre faisable.

3.4 Recherche tabou

La recherche tabou commence à partir d'une solution infaisable et tente à partir de celle-ci de trouver une solution faisable. La fonction à minimiser est la somme des violations des contraintes. Nous utilisons un seul voisinage, mais deux stratégies d'exploration et de sélection de voisins différentes. Dans la suite, nous détaillons notre opérateur de voisinage, la méthodologie tabou, et l'exploration du voisinage.

Opérateur de voisinage and évaluation de voisins L'opérateur de voisinage dans notre recherche tabou déplace une VNF d'un serveur à un autre. Soit *sol* la solution courante ; un mouvement est représenté par un couple $\langle g, f \rangle$ et est valide si $g \in G^{\Xi}_{\texttt{VNF}}$, $f \in F_{\texttt{SERV}}$ et $\mathtt{s}_{sol}(g) \neq f$. La solution obtenue par l'exécution de $\langle g, f \rangle$ sur la solution *sol* est notée $sol[g, f]$. Les solutions sont évaluées en fonction de leur violation des contraintes $v(sol)$. Si $v(sol) > v(sol[g, f])$ alors le mouvement $\langle g, f \rangle$ améliore la solution *sol*.

Liste tabou et critère d'aspiration Un mouvement $\langle g, f \rangle$ est considéré comme tabou si son noeud VNF g est actuellement fixé à tabou. Un noeud VNF g est tabou pour les prochaines θ itérations après l'exécution d'un mouvement $\langle g, f \rangle$. Le statut tabou d'un mouvement $\langle g, f \rangle$ est ignoré si $v(sol[g, f]) < v(sol^*)$ où sol^* est la meilleure solution connue jusqu'ici.

Exploration de voisinages Au cours de la recherche tabou, nous ne considérons qu'un type d'opérateur de voisinage, la réaffectation d'une VNF à un autre serveur. Le voisinage est calculé sur base de la solution courante *sol* et l'ensemble des chemins dirigés P sur le graphe du data center. Nous utilisons deux stratégies d'exploration différentes sur ce même voisinage : *CMBExploration* et *LAExploration*. Ces stratégies sont conçues pour minimiser chacune les violations d'une catégorie spécifique de contraintes.

CMBExploration. Cette stratégie est utilisée pour minimiser les violations combinées des contraintes de demande de coeurs CPU, de mémoire et de bande passante (contraintes CMB, éq.1, 2 et 4). De toutes les VNFs, nous sélectionnons la VNF g qui contribue le plus à la violation des contraintes CMB. Nous sélectionnons alors le serveur f (différent de celui auquel la VNF est assignée dans la solution courante) résultant dans le moins de violations des contraintes CMB. Si le mouvement $\langle g, f \rangle$ est tabou (i.e. le noeud VNF est tabou) et si le critère d'aspiration n'est pas rempli alors la meilleure VNF non-taboue g' et le serveur f' correspondant sont choisis.

LAExploration. Cette stratégie est utilisée pour minimiser les violations de la contrainte de latence (éq. 4). Tout d'abord un des chemins des SRs (dans l'ensemble T^{Ξ}) avec la plus haute violation de la contrainte de latence est choisi. Ensuite, nous créons un ensemble *Cand* de tous les VNFs figurant sur ce chemin. Pour chaque $g \in Cand$ nous déterminons le meilleur serveur, c'est-à-dire le serveur $f \in F_{\texttt{SERV}}$ résultant dans le moins de violations lorsqu'il est attribué à g. Cela nous donne un ensemble de mouvements, à par-

tir duquel nous sélectionnons le meilleur mouvement non-tabou (ou mouvement tabou respectant le critère d'aspiration).

3.5 Algorithme complet

Nous avons maintenant tous les éléments pour mettre ensemble l'algorithme utilisé pour résoudre le problème hors ligne, cf. algorithme 2. On commence par le calcul de l'ensemble des chemins dirigés entre chaque paire de noeuds de serveur et chaque paire de noeuds serveur et I/O-switch dans le data center. Ensuite, une solution initiale est créée sur base de ces chemins, en utilisant l'heuristique round-robin gloutonne. Cette solution, si infaisable, est améliorée par la recherche tabou. Si les contraintes CMB ne sont pas respectées, le voisinage est exploré avec *CMBExploration*, le mouvement sélectionné est exécuté et la VNF correspondante est fixée tabou. Si les contraintes CMB sont respectées, alors les violations doivent provenir de la contrainte de latence et la stratégie d'exploration correspondante est utilisée. Si la nouvelle solution améliore la meilleure solution connue (la nouvelle solution a moins de violations de contraintes totales que la meilleure connue), alors la meilleure solution connue est mise à jour.
La recherche tabou se termine lorsqu'une solution faisable a été atteinte ou lorsqu'une limite de temps ou un nombre maximal d'itérations est atteint.

Algo. 2: Résolution de la variante hors ligne du PISR

Output : Une solution sol^*
1 $\texttt{tabu} \leftarrow \emptyset$; $sol, sol^* \leftarrow \perp$; $\texttt{it} \leftarrow 0$;
2 $P \leftarrow$ précalcul des chemins sur le graphe $\mathscr{D}$;
3 $sol, sol^* \leftarrow$ création d'une solution intiale avec heuristique gloutonne round-robin sur base de P;
4 **while** $\neg\texttt{timeout} \wedge \texttt{it} < maxit \wedge v(sol^*) > 0$ **do**
5 **if** $v_{cpu}(sol) + v_{mem}(sol) + v_{bp}(sol) > 0$ **then**
6 choisir mouv. $\langle g, f \rangle$ avec *CMBExploration*;
7 $sol \leftarrow sol[g, f]$; $\texttt{tabu}[g] \leftarrow \texttt{it} + \theta$;
8 **else**
9 choisir mouv. $\langle g, f \rangle$ avec *LAExploration*;
10 $sol \leftarrow sol[g, f]$; $\texttt{tabu}[g] \leftarrow \texttt{it} + \theta$;
11 **end**
12 **if** $v(sol) < v(sol^*)$ **then** $sol^* \leftarrow sol$
13 **end**
14 **return** sol^*;

4 Méthodologie pour problème en ligne

Dans cette section nous montrons comment adapter les méthodes présentées pour la variante hors ligne à la variante en ligne du PISR. Dans la variante en ligne l'ensemble Ξ de SRs à intégrer dans le data center n'est pas connu à l'avance. Les demandes d'intégration de nouveaux SRs apparaissent au fil du temps au data center. Le système doit alors décider d'accepter ou de rejeter chaque demande. Pour ce faire, un nouveau problème de satisfaction de contraintes doit être résolu pour chaque nouvelle demande de SR. Ce nouveau problème correspond alors au mapping des noeuds dans le nouveau SR aux noeuds du data center de manière à respecter les contraintes du problème, tenant compte des SRs déjà incorporés dans le data center.
Le problème en ligne peut donc être considéré comme une séquence de n problèmes de satisfaction de contraintes, où n est le nombre total de SRs à allouer à jamais. Chaque problème i dans cette séquence est défini par $\mathscr{D} = (F, A)$ le graphe du data center, le singleton du SR à allouer $\Xi_i = \{\mathscr{S}^i\}$ avec $\mathscr{S}^i = (G^i, H^i, T^i)$, Ψ_i l'ensemble des SRs en cours d'exécution sur le data center et la solution courante sol_{i-1}. La solution sol_{i-1} est faisable et correspond au mapping entre les noeuds de G^{Ψ_i} et les nœuds du data center et au mapping (sur la base de l'ensemble précalculé de chemins P) entre les arcs H^{Ψ_i} et les chemins dirigés sur le data center. Notez que $\Psi_0 = \emptyset$.
La méthodologie globale utilisée pour le problème en ligne est la suivante : à l'initialisation les chemins dirigés sur le data center sont précalculés et une solution vide *sol* est créée. Le système reste alors à l'écoute de nouvelles demandes de SRs. Chaque fois qu'une nouvelle demande de SR $\mathscr{S}$ arrive, une tentative de mise à jour de la solution courante *sol* par rapport au nouveau SR est faite. Cette mise à jour consiste à appliquer une heuristique de génération de solution initiale et puis la recherche tabou. Si une solution sol' faisable peut être trouvée sur base de *sol* alors l'intégration du nouveau SR $\mathscr{S}$ est acceptée et sol' devient la nouvelle solution courante. Dans le cas contraire, le SR $\mathscr{S}$ est rejeté par le système.
L'adaptation de la recherche tabou décrite dans la section 3.4 à la variante en ligne est simple. Il suffit, pour tout problème i, d'interdire à la recherche de bouger une VNF g telle que $g \notin G^i$ vers un serveur différent (avec $\mathscr{S}^i = (G^i, H^i, T^i)$ le nouveau SR).

4.1 Solution initiale avec heuristiques alternatives

Dans le contexte du problème en ligne, la création d'une solution initiale consiste à prendre une solution faisable sol_{i-1} et à la transformer en une solution sol_i éventuellement infaisable (à améliorer par la recherche tabou). Cette transformation ne peut cependant pas modifier le mapping décrit par la solution sol_{i-1}, uniquement l'étendre au nouveau SR $\mathscr{S}^i = (G^i, H^i, T^i)$.
La méthode round-robin gloutonne peut être utilisée telle quelle. Nous considérons également deux heuristiques de génération de solutions initiales supplémentaires. Chacune commence par le placement aléatoire des noeuds de source et de puit du SR sur les noeuds I/O-switch du graphe du data center. Leur comportement ultérieur est décrit ci-dessous.

Aléatoire Pour chaque VNF $g \in G^i_{\texttt{VNF}}$ nous essayons d'abord de trouver un noeud serveur $f \in F_{\texttt{SERV}}$, tels que le mapping entre g et f ne donne pas lieu à des violations

des contraintes du problème. Si plusieurs serveurs sont possibles, l'un d'eux est choisi aléatoirement. Si aucun tel serveur n'est disponible, un serveur est choisi au hasard. Enfin, la solution actuelle sol_{i-1} est mise à jour, attribuant le serveur f à la VNF g.

Aléatoire avec Minimisation #Serveurs Pour chaque VNF $g \in G^i_{\texttt{VNF}}$ nous essayons d'abord de sélectionner un serveur $f \in F_{\texttt{SERV}}$ tel que l'affectation de g à f ne donne pas lieu à des violations de contraintes *et* tel que le nombre de serveurs utilisés est minimal. Si aucun tel serveur ne peut être trouvé, nous utilisons le serveur qui provoque le moins de violations quand alloué à g.

5 Résultats expérimentaux

Dans cette section, nous décrivons d'abord le dispositif expérimental utilisé pour nos expériences de calcul. Ensuite, les résultats pour la variante hors ligne et la variante en ligne sont présentés.

5.1 Configuration expérimentale

Notre approche a été implémentée en tant que recherche locale basée sur les contraintes en Comet. Toutes les expériences ont été effectuées sur des machines virtuelles Xen tournant sur un coeur d'un CPU Intel Core2 Quad Q6600@2.40GHz avec 1Go de RAM. La longueur de la liste tabou θ est fixée à 10. Pour la configuration expérimentale nous devons simuler à la fois un data center et un ensemble de SRs.

Simulation d'un data center Nous considérons à la fois un petit et un grand data center. Le petit comprend 2 switches centraux, 4 switches intermédiaires, 8 ToR-switches et 32 serveurs. Chaque switch central est en outre relié à 2 I/O-switches. Chaque serveur dispose de 8 coeurs de CPU et de 2048Mo de mémoire. La bande passante interne des serveurs et entre des paires de switches est supposée être de 10 Gbps, tandis que la bande passante sur les liens entre les serveurs et le ToR-switch (dans les deux sens) est supposé être 1Gbps. La latence sur chaque lien (dans les deux sens) est supposé prendre la valeur 1.
Le grand data center comprend 32 switches centraux, 32 switches intermédiaires, 500 Tor-switches et 10 000 serveurs. Chaque switch central est en outre relié à 5 I/O-switches. La configuration restante est le même que pour le petit data center.

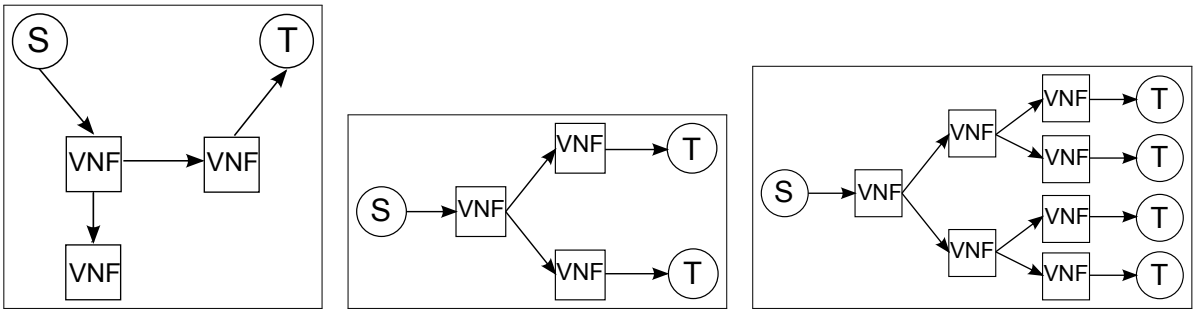
Simulation de SRs Nous considérons quatre modèles de SR différents. Les propriétés de chacun sont détaillés dans la Table 1 et leurs topologies sont illustrées dans les Figures 1a et 2. Nous supposons que les paquets envoyés entre les composantes des SRs ont une taille de 1024 bits.
A partir des modèles de SR et sur base des propriétés dans la Table 1, nous générons des ensembles de SRs aléatoirement. La taille des ensembles que nous générons dépend des expériences spécifiques et sera détaillée dans les sections pertinentes.

5.2 Variante hors ligne

Pour le problème hors ligne nous avons considéré le petit et le grand data center. Dans la suite nous décrivons brièvement les résultats (moyennes arrondies sur 10 runs indépendants) obtenus sous une limite de temps d'exécution de 3600 secondes CPU et une limite de 10^6 itérations pour la recherche tabou. Aucune limite de temps n'est imposée à la construction de la solution initiale. Les résultats détaillés sont disponibles sur `http://becool.info.ucl.ac.be/resources/PISR`.

Petit data center Pour le petit data center nous générons $k \in \{8,9,10\}$ SRs de chaque type. Au total, nous avons donc $\{32,36,40\}$ SRs. En ce qui concerne la latence maximale z_h sur les arcs $h \in H^{\Xi}$, nous supposons que tous les arcs ont la même limite z. Pour les expériences nous considérons des valeurs différentes de $z \in \{10,11,12,13,14,15,16,17,18,19,20\}$. Les instances sont dénotées small-NSX, avec X le nombre de SRs. Les expériences montrent que la valeur de la latence maximale a un grand impact sur les résultats. Pour les instances small-NS32 et small-NS36 aucune solution faisable ne peut être trouvée pour des latences maximales $z \leq 13$, alors que pour, par exemple, small-NS32 avec $14 \leq z \leq 20$ l'heuristique round-robin gloutonne est capable de trouver une solution faisable en 0.9 secondes, ce qui rend l'exécution de la recherche tabou inutile. Pour l'instance small-NS36 notre heuristique gloutonne n'est jamais en mesure de trouver une solution faisable initiale, mais la recherche tabou améliore la solution initiale jusqu'à ce qu'une solution faisable se trouve après 2.1 secondes ($14 \leq z \leq 20$). Enfin aucune solution faisable ne peut être trouvée pour les instances small-NS40 ($10 \leq z \leq 20$). Cependant, la recherche tabou est en mesure de réduire le nombre total de violations avant que le temps limite ne soit atteint. Notez que lorsque $z = 20$ notre recherche est capable de trouver des solutions qui ne violent ni la contrainte sur la bande passante ni la contrainte de latence pour cette instance. L'infaisabilité de l'instance small-NS40 découle ainsi des ressources des coeurs de CPU et de mémoire insuffisantes dans le petit data center.

Grand data center Nous générons $k \in \{1000,1500,2000,2500,3000\}$ SRs de chaque type. Au total, nous avons donc $\{4000,6000,8000,10000,12000\}$ SRs pour le grand data center. En ce qui concerne la latence maximale z_h pour les arcs $h \in H^{\Xi}$, nous supposons que tous les arcs ont la même limite z. Pour les expériences nous considérons à nouveau des valeurs de 10 à 20 pour la latence maximale z. Les instances sont dénotées large-NSX avec X le nombre de SRs. Pour les instances large-NS12000 ni l'heuristique round-roubin gloutonne, ni la recherche tabou n'arrivent à une solution faisable pour



(a) Contrôle d'accès au réseau (b) Mise en tunnel (c) Répartition de charge

FIGURE 2 – Modèles de SRs

TABLE 1 – Description de modèles de SRs

Nom	#I/O	#VNF	Coeurs CPU per VNF	Mémoire (MB) par VNF	Traffic (pps) par paire de VNFs
Détection d'intrusions	2	5	$\{0.5, 2\}$	100..1024	38800..100000
Répartition de charge	5	7	$\{0.7, 1, 1.5\}$	100..200	25000..100000
Contrôle d'accès au réseau	2	3	$\{0.5, 1\}$	100..200	96000..100000
Mise en tunnel	3	3	$\{1.5, 2.5\}$	200..1024	50000..100000

tout z. Comme pour l'instance small-NS40, l'infaisabilité semble provenir du nombre limité de coeurs CPU et de la mémoire limitée. Pour les instances large-NS4000 à large-NS10000 l'heuristique gloutonne est capable de trouver des solutions faisables initiales pour tous les cas où $z \geq 16$. Toutefois, aucune solution faisable ne peut être trouvée pour $10 \leq z \leq 11$. Dans l'ensemble, le temps d'exécution maximal nécessaire pour arriver à une solution faisable varie entre 196 (NS6000 avec $z = 12$) et 839 (NS8000 avec $z = 12$) secondes.

5.3 Variante en ligne

Pour le problème en ligne, nous considérons plusieurs configurations de l'approche décrite. Les configurations diffèrent dans l'algorithme utilisé pour produire la solution initiale, et dans le fait d'utiliser la recherche tabou pour améliorer cette solution initiale ou non. Les différentes configurations sont les suivantes :

- `A` – génération Aléatoire de la solution initiale
- `RR` – génération de la solution intiale avec l'heuristique Round-Robin gloutonne
- `AMinS` – génération Aléatoire de la solution initialee avec Minimisation #Serveurs
- `X+TS` – génération de la solution initiale avec la méthode X avec recherche Tabou consécutive

Pour l'expérience en ligne nous réutilisons le grand data center et considérons une instance large-NS16000 avec 16000 SRs, de nouveau sous différentes valeurs de latence maximale. Nous supposons que les SRs, une fois intégrés dans le data center, continuent à fonctionner à l'infini. L'ordre dans lequel arrivent les demandes de SRs a été généré aléatoirement. Pour chaque nouveau SR nous essayons d'abord de trouver une solution faisable avec l'heuristique de génération de solutions initiales, puis, si l'heuristique produit une solution infaisable et que la configuration actuelle comprend la recherche tabou, nous permettons 30 secondes et 10^6 itérations à la recherche tabou afin de trouver une solution faisable. Si après écoulement du temps ou des itérations limites aucune attribution faisable n'a été trouvée, la demande est rejetée. Une nouvelle demande est traitée dès que la dernière a été acceptée ou rejetée.

Les différentes configurations sont comparées en fonction de deux critères : le nombre de séquence du premier SR à être rejeté (à noter que quand un SR est rejeté, le processus global continue à fonctionner et reste à l'écoute de nouvelles demandes entrantes qui pourraient être acceptées ou rejetées) et le nombre total de SRs acceptés. Bien sûr, nous préférons que le numéro de séquence du premier SR rejeté et le nombre total de SRs acceptés soient les plus grands possible. Ces deux mesures sont données dans la Table 2. Les colonnes indiquent pour chacune des six configurations considérées le numéro de séquence du premier SR rejeté et le nombre total de SRs acceptés sous forme : *premier rejeté - total acceptés*. Chaque ligne correspond à l'instance large-NS16000 avec une latence maximale z différente.

En termes de premier SR rejeté, nous voyons que quand $z < 15$, la configuration `RR+TS` est la meilleure : avec $z = 11$, `RR+TS` est capable d'affecter 4667 SRs, tandis que la configuration `RR` peut affecter seulement les 326 premiers SRs entrants et les autres configurations qu'un seul. Avec $z > 15$ c'est la configuration `AMinS+TS` qui donne les meilleurs résultats en termes premier SR rejeté.

TABLE 2 – Numéro de séquence du premier SR rejeté (meilleur *en italique*) et nombre total de SRs acceptés (meilleur **en gras**) sur l'instance large-NS16000

z	A	A+TS	RR	RR+TS	AMinS	AMinS+TS
10	*2* - 9963	*2* - 9973	*2* - 10777	*2* - 10917	*2* - 10943	*2* - **11153**
11	2 - 9973	2 - 9994	327 - 11898	*4668* - **12480**	2 - 11133	2 - 11686
12	2 - 9959	2 - 9968	429 - **12588**	*2938* - 12503	4 - 11139	2 - 11718
13	2 - 10008	2 - 12515	429 - **12627**	*10173* - 12493	2 - 11197	8 - 11871
14	2 - 10011	2 - **12678**	429 - 12622	*10173* - 12495	2 - 11208	34 - 11887
15	1652 - 11899	5068 - **13208**	*10134* - 12623	*10134* - 12515	75 - 11514	177 - 11703
16	10257 - **13144**	10245 - 13141	10134 - 12506	10134 - 12506	6431 - 11640	*11651* - 11653
17	10293 - **13160**	10280 - 13157	10134 - 12506	10134 - 12506	7475 - 11632	*11655* - 11656
18	10284 - **13159**	10264 - 13151	10134 - 12505	10134 - 12505	11647 - 11647	*11659* - 11660
19	10265 - **13155**	10253 - 13134	10134 - 12505	10134 - 12505	11640 - 11643	*11653* - 11656
20	10286 - **13161**	10249 - 13143	10134 - 12506	10134 - 12506	*11635* - 11636	11632 - 11631

Toutes les configurations sont en mesure d'affecter au moins 9958 des SRs, indépendamment de z. En termes de nombre total de SRs affectés, la configuration A trouve de meilleurs résultats que les autres dans 5 des 11 cas tandis que A+TS trouve de meilleurs résultats que les autres dans 2 cas seulement. Chacune des configurations RR+TS et AMinS+TS trouve de meilleurs résultats que les autres dans un des 11 cas, et la configuration RR trouve de meilleurs résultats que les autres dans 2 sur 11 cas.
Une autre observation que nous faisons est que les procédures d'initialisation (A, RR et AMinS) ne sont pas toujours pires que leur combinaison avec la recherche tabou (Rand+TS, RR+TS, MinS+TS) en termes de nombre total de SRs acceptés. Cela peut être dû au fait que les procédures de génération de solution initiale écartent un SR plus facilement que les procédures combinées. Il est possible que le rejet d'un SR permette d'affecter plusieurs des SRs suivants, ce qui n'aurait pas été possible autrement, si le premier SR aurait été accepté. C'est pour cette raison aussi qu'une configuration donnée ne sera pas toujours en mesure d'affecter un nombre total plus élevé de SRs lorsque la latence maximale augmente. Le même comportement peut être observé en termes de premier SR rejeté, ici les variations sont probablement dues à des décisions aléatoires dans notre procédure.

6 Travaux apparentés

Le PISR peut être considéré comme un cas particulier du problème d'intégration de réseaux virtualisés (Virtual Network Embedding Problem, VNEP). Dans le VNEP un ensemble de réseaux virtuels (VNs), décrits par des graphes non orientés, doit être incorporé dans un réseau physique, représenté par le graphe substrat. Les demandes d'intégration de nouveaux réseaux virtuels arrivent au fil du temps. Des noeuds VN doivent être mappés sur les nœuds du graphe substrat, tandis que les liens VN sont mappés sur des chemins dans le graphe substrat, les deux de manière à respecter un ensemble de contraintes, comme par exemple des contraintes de ressources. Les VN ont un temps de vie associé, après lequel ils disparaissent du réseau physique. Le VNEP demande des solutions optimisant une fonction, correspondant généralement à un équilibre entre les coûts et les recettes ; notre PISR en revanche est un problème de satisfaction de contraintes pur. En outre, le VNEP ne considère pas explicitement le graphe substrat comme un data center, tandis que le NSEP est explicitement défini sur celui-ci. Notre heuristique round-robin gloutonne vise spécifiquement l'intégration sur une structure de data center. Comme nous envisageons des data centers, notre graphe correspondant au réseau physique (jusqu'à 10^4 noeuds serveur) est considérablement plus grand que les graphes substrat généralement considérés dans la littérature VNEP, qui ont jusqu'à 100 noeuds. Différentes techniques d'optimisation ont été appliquées au VNEP dans le passé, comme des heuristiques basées sur la programmation linéraire (e.g. [1]) et branch-and-bound (e.g. [11]), recherches avec retours en arrière (e.g. [12]), des heuristiques gloutonnes (e.g. [17]) et des métaheuristiques (e.g. [5]). Des travaux récents ont également examiné le VNEP sous l'aspect de défaillances dans le réseau physique sous-jacent ([15]). Un état-de-l'art complet pour le VNEP peut être trouvé dans [6].
Un certain nombre de problèmes liés à l'allocation de ressources des data centers sont également traités dans la littérature. Dans ces scénarios, les contraintes de bande passante sont généralement plus restrictives que dans notre PISR. Dans [8], les auteurs considèrent le problème de l'intégration d'un ensemble de machines virtuelles sur un data center de manière à satisfaire des contraintes de bande passante sur les liens et de capacité sur les serveurs (coeurs CPU, mémoire, et disque dur). Chaque serveur est associé à une machine virtuelle et aucune contrainte de latence n'est considérée. Chaque nouvelle demande est constituée de milliers de machines virtuelles avec une matrice décrivant la bande passante nécessaire entre ces machines. Les serveurs du data center physique sont préconfigurées en clusters et toutes les machines virtuelles d'une même demande seront attribuées sur les serveurs d'un même cluster. Le problème de placement de machines virtuelles a également été pris en compte dans [14]. Dans cet article le scénario est simple : nous devons trouver un mapping un-à-un de n machines virtuelles à n slots minimisant le trafic total sur le data center. Les auteurs proposent une approche heu-

ristique basée sur du clustering. Le problème du placement de machines virtuelles sur des data centers pris en compte dans [13] vise à minimiser l'énergie consommée sur les switches et les liens. Un mapping un-à-un des machines virtuelles vers des serveurs satisfaisant les contraintes sur la capacité de liens et des serveurs doit être trouvé. Ceci est fait en utilisant une heuristique gloutonne. Enfin, dans [10], la recherche locale basée sur les contraintes est utilisée pour intégrer efficacement des réseaux locaux virtuels dans des data centers.

7 Conclusion

Dans cet article, nous considérons un problème de mapping issu du contexte de la virtualisation des fonctions réseau, une approche pour le déploiement de nouveaux services dans des réseaux de taille importante. Des fonctions de réseau virtualisées et regroupées dans des services réseau (SRs) sont exécutées sur des serveurs standards à haute performance, généralement situés dans des data centers. Les différentes ressources du data center doivent alors être partagées entre un ensemble de SRs. Un data center peut être vu comme un graphe orienté avec boucles dans lequel un ensemble de graphes orientés acycliques, chacun correspondant à un SR, doit être incorporé. Le mapping produit doit respecter un ensemble de contraintes de ressources et de performance. Nous proposons plusieurs heuristiques de génération de solution initiale ainsi qu'une recherche tabou. Nous montrons comment l'approche proposée peut être facilement adaptée à la variante en ligne du problème. Les résultats expérimentaux montrent que notre approche est viable et fonctionne bien, même pour les grands data centers où un grand nombre de SRs doit être assigné. Les travaux futurs vont suivre deux directions. Tout d'abord, nous voulons nous concentrer sur l'intégration du calcul de chemins dirigés (sur le data center) dans les heuristiques et dans la recherche locale. Il n'y aurait plus de phase de précalcul et la modification des chemins serait autorisée dans la recherche tabou. Deuxièmement, nous voulons améliorer la robustesse de notre approche en l'adaptant à un grand nombre de scénarios, en tenant compte de différentes distributions de SRs arrivants et aussi de la fin de vie des SRs.

Références

[1] M. Chowdhury, M. R. Rahman, and R. Boutaba. ViNEYard virtual network embedding algorithms with coordinated node and link mapping. *IEEE/ACM Transactions on Networking (TON)*, 20(1), 2012.

[2] N. M. K. Chowdhury and R. Boutaba. A survey of network virtualization. *Computer Networks*, 54(5), 2010.

[3] M. Dobrescu, N. Egi, K. Argyraki, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy. Routebricks exploiting parallelism to scale software routers. In *Proceedings of the ACM SOSP '09*. ACM, 2009.

[4] ETSI NFV ISG. Network functions virtualisation, white paper, 2012.

[5] I. Fajjari, N. Aitsaadi, G. Pujolle, and H. Zimmermann. VNE-AC virtual network embedding algorithm based on ant colony metaheuristic. In *Proccedings of IEEE ICC 2011*, 2011.

[6] A. Fischer, J. F. Botero, M. T. Beck, H. de Meer, and X. Hesselbach. Virtual network embedding a survey. *IEEE Communications Surveys Tutorials*, 15(4), 2013.

[7] A. Greenhalgh, F. Huici, M. Hoerdt, P. Papadimitriou, M. Handley, and L. Mathy. Flow processing and the rise of commodity network hardware. *SIGCOMM Comput. Commun. Rev.*, 39(2), Mar. 2009.

[8] C. Guo, G. Lu, H. Wang, S. Yang, C. Kong, P. Sun, W. Wu, and Y. Zhang. Secondnet a data center network virtualization architecture with bandwidth guarantees. In *ACM CONEXT 2010*, 2010.

[9] J. R. Hamilton. On designing and deploying internet-scale services. In *LISA*, volume 7, 2007.

[10] T.-V. Ho, Y. Deville, O. Bonaventure, and P. Francois. Traffic engineering for multiple spanning tree protocol in large data centers. In *23rd International Teletraffic Congress*. IEEE Explorer, 2011.

[11] J. Inführ and G. R. Raidl. Introducing the virtual network mapping problem with delay, routing and location constraints. In J. Pahl, T. Reiners, and S. Voß, editors, *Network Optimization*, number 6701 in LNCS. Springer Berlin Heidelberg, 2011.

[12] J. Lischka and H. Karl. A virtual network mapping algorithm based on subgraph isomorphism detection. In *Proceedings of the ACM VISA 2009*, 2009.

[13] V. Mann, A. Kumar, P. Dutta, and S. Kalyanaraman. Vmflow leveraging vm mobility to reduce network power costs in data centers. *Networking*, 6640, 2011.

[14] X. Meng, V. Pappas, and L. Zhang. Improving the scalability of data center networks with traffic-aware virtual machine placement. In *INFOCOM proceedings*. IEEE, 2010.

[15] M. R. Rahman, I. Aib, and R. Boutaba. Survivable virtual network embedding. In *NETWORKING 2010*. Springer, 2010.

[16] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. Sekar. Making middleboxes someone else's problem network processing as a cloud service. In *Proceedings of ACM SIGCOMM 2012*, SIGCOMM '12. ACM, 2012.

[17] Y. Zhu and M. H. Ammar. Algorithms for assigning substrate network resources to virtual network components. In *INFOCOM*, 2006.

Autours des Stratégies de Réduction de la Base de Clauses Apprises

Saïd Jabbour[1] **and Jerry Lonlac**[1,2] **and Lakhdar Saïs**[1] **and Yakoub Salhi**[1]

[1] CRIL - CNRS, Université d'Artois, Lens, France F-62307 Cedex
[2] Département d'Informatique - Université de Yaoundé 1 B.P. 812 Yaoundé, Cameroun
{jabbour, lonlac, sais, salhi}@cril.fr

Résumé

Dans cet article, nous revisitons un point important des solveurs SAT de type CDCL, à savoir les stratégies de gestion de la base des clauses apprises. Notre motivation prend sa source d'une simple observation sur les performances remarquables de deux stratégies simples : élimination des clauses de manière aléatoire et celle utilisant la taille des clauses comme critère pour juger de la pertinence d'une clause apprise. Nous proposons d'abord une stratégie de réduction, appelée "Size-Bounded Randomized strategy" (SBR), qui combine le maintien de clauses courtes (de taille bornée par k), tout en supprimant aléatoirement les clauses de longueurs supérieures à k. La stratégie résultante surpasse celles de état-de-l'art, à savoir la stratégie LBD (Literal Block Distance), sur les instances SAT de la dernière compétition internationale SAT 2013. Renforcé par l'intérêt de garder les clauses courtes, nous proposons plusieurs nouvelles variantes dynamiques, et nous discutons de leurs performances.

Abstract

In this paper, we revisit an important issue of CDCL-based SAT solvers, namely the learned clauses database management policies. Our motivation takes its source from a simple observation on the remarkable performances of both random and size-bounded reduction strategies. We first derive a simple reduction strategy, called Size-Bounded Randomized strategy (in short SBR), that combines maintaing short clauses (of size bounded by k), while deleting randomly clauses of size greater than k. The resulting strategy outperform the state-of-the-art, namely the LBD based one, on SAT instances taken from the last SAT competition. Reinforced by the interest of keeping short clauses, we propose several new dynamic variants, and we discuss their performances.

1 Introduction

Aujourd'hui, le problème SAT a gagné une audience considérable avec l'avénement d'une nouvelle génération de solveurs capables de résoudre des grandes instances issues du codage des applications du monde réel ainsi que par le fait que ces solveurs constituent d'importants composants de base pour plusieurs domaines, e.g., SMT (SAT Modulo Théorie), démonstration automatique, comptage de modèles, problème QBF, etc. Ces solveurs communément appelés solveurs SAT modernes CDCL (Conflict Driven, Clause Learning) [30, 16] sont basés sur la propagation unitaire classique [14] finement combinée avec des structures de données efficaces (ex. Watched literals), des politiques de redémarrages [23, 20, 26], l'heuristique de choix de variables basée sur les activités "VSIDS" (Variable State Independant, Decading Sum) [30], ainsi que l'apprentissage de clauses [28, 30, 37]. L'apprentissage de clauses est actuellement reconnu comme le composant le plus important des solveurs SAT modernes. L'idée principale est que lorsqu'une branche courante de l'arbre de recherche conduit à un conflit, l'apprentissage de clauses vise à dériver une clause qui succinctement exprime les causes du conflit. Une telle clause apprise est ensuite utilisée pour élaguer l'espace de recherche. L'apprentissage de clauses est également connu dans la littérature sous le nom "Conflict Driven Clause Learning" (CDCL) faisant référence actuellement au schéma d'apprentissage premier UIP le plus connu et le plus utilisé, qui est d'abord intégré dans les solveurs SAT Grasp [36] et efficacement mis en œuvre dans zChaff [31]. La plupart des solveurs SAT, intégrent ce système d'apprentissage de clauses. Théoriquement, en intégrant l'apprentissage de clauses à la procédure DPLL classique [13], le solveur SAT ob-

tenu formulé comme un système de preuve est aussi puissant que la résolution générale [33, 35]. Dans la pratique, l'efficacité des solveurs de type CDCL dépend fortement de la stratégie utilisée pour gérer la base des clauses apprises. En effet, comme à chaque conflit une nouvelle clause est ajoutée à la base des clauses apprises, la taille de la base croît de manière exponentielle. Pour éviter cette explosion combinatoire, plusieurs stratégies de gestion de la base des clauses apprises ont été proposées (nous pouvons citer [30, 16, 5, 4, 21]). Ces stratégies visent à maintenir une base de clauses apprises de taille raisonnable en éliminant les clauses jugées non-pertinentes pour la suite de la recherche. Toutes ces stratégies de gestion suivent une séquence temporelle de nettoyage prédéfinie où l'intervalle entre deux étapes successives de réduction est plus ou moins important. A chaque conflit, une activité est associée à la clause apprise (stratégie statique). Une telle heuristique basée sur l'activité vise à pondérer chaque clause en fonction de sa pertinence dans le processus de recherche. Dans le cas des stratégies dynamiques, ces activités sont mises à jour dynamiquement. La réduction de la base des clauses apprises consiste à supprimer les clauses inactives ou non-pertinentes. Même si toutes les stratégies d'élimination des clauses apprises proposées s'avèrent empiriquement efficaces, déterminer la clause la plus pertinente pour le processus de recherche reste un véritable défi. Il est important de noter que l'efficacité de la plupart des stratégies de gestion des clauses apprises de l'état de l'art dépend fortement de la fréquence de nettoyage et de la quantité de clauses à supprimer à chaque nettoyage.

Différentes implémentations de solveurs SATsont proposées chaque année à la compétition SAT, ils incluent plusieurs améliorations des principales composantes des solveurs CDCL. Les compétitions SATstimulent le développement des implémentations efficaces. Cependant, une telle course à la mise en œuvre la plus efficace a conduit à rendre les solveurs de plus en plus complexes et dépendants de nombreux paramètres. Ces paramètres sont soient statiques (fixés avant la recherche), soit dynamiques, leurs valeurs sont conditionnellement définies au cours de la recherche en fonction du comportement des solveurs lors de la recherche. Ces implémentations sophistiquées augmentent la difficulté à comprendre ce qui est essentiel de ce qui ne l'est pas. Dans [25], une analyse empirique portant sur les principales techniques qui ont contribué aux performances impressionnantes des solveurs SAT modernes a été menée. Ceci peut être vu comme un premier pas vers une compréhension profonde des solveurs SAT modernes.

Dans ce papier, nous réexaminons une question importante des solveurs SAT de type CDCL, à savoir les stratégies de gestion de la base des clauses apprises. Il est *important de noter* à ce point que, les stratégies qui considèrent les clauses courtes comme les plus pertinentes ("size-bounded based reduction strategies") ont été proposées depuis 1996 par Marques Silva et Sakallah (Grasp [29]), Bayardo et Schrag (RelSAT [8]). La plupart des solveurs SAT de l'état de l'art gardent systématiquement les clauses binaires, tandis que pour les clauses de taille supérieure à deux, plusieurs mesures sophistiquées de pertinence ont été proposées pour prédire la qualité des clauses.

Notre motivation pour ce travail prend sa source d'une simple observation sur les performances remarquables des stratégies de réduction basées sur la mise en place d'une borne supérieure sur la taille des clauses apprises. A partir de ces premiers résultats, nous avons décidé d'examiner d'autres stratégies "naïves" telles que la stratégie de gestion aléatoire et celle utilisant une file (First In First Out). Le but est de quantifier l'écart en termes de performance entre ces stratégies simples et celles mises en œuvre dans les solveurs SAT de l'état de l'art. Ensuite, nous dérivons une stratégie de réduction, appelée "Size-Bounded Randomized Strategy" (abrégée SBR), qui conserve les clauses de petites tailles (de taille inférieure ou égale à k), tout en supprimant aléatoirement les clauses de taille supérieure à k. Renforcé par l'intérêt de garder les clauses courtes, nous proposons plusieurs nouvelles mesures dynamiques qui nous permettent de quantifier la pertinence d'une clause apprise donnée en fonction de l'état courant du processus de recherche. Intuitivement, *une clause apprise de petite taille avec les littéraux assignés le plus souvent au en haut de la branche courante de l'arbre de recherche est considérée comme plus pertinente*. Plusieurs stratégies basées sur la pertinence sont ensuite dérivées, nous permettant de garder les clauses apprises qui sont plus susceptibles de couper des branches en haut de l'arbre de recherche. Toutes ces stratégies sont intégrées dans le solveur Minisat 2.2 et comparées aux solveurs SAT de l'état de l'art sur les instances d'applications prises de la dernière compétition SAT 2013. Pour confirmer la supériorité des mesures basées sur la taille contre celles basées sur le LBD, nous présentons également les résultats obtenus en substituant LBD avec une activité dynamique basée sur la taille dans le solveur SAT *Glucose* 3.0. Les résultats obtenus remettent au goût du jour les anciennes stratégies basées sur la taille des clauses, proposées il y a plus d'une quinzaine d'années [29, 7, 8] (voir Section 2 - Travaux Connexes). Ces résultats ne sont pas surprenants, puisque les petites clauses sont généralement préférées pour leur capacité à réduire encore plus l'espace de recherche. C'est aussi pourquoi tous

les solveurs SAT gardent systématiquement les clauses binaires apprises.

Ce papier est organisé comme suit. Nous rappelons tout d'abord les travaux connexes dans la section 2, ensuite après quelques notations, définitions préliminaires, et notions de bases autours des solveurs SAT, nous motivons notre étude en examinant les stratégies de suppressions basées sur la limitation de la taille, sur un choix aléatoire des clauses à supprimer, et sur une gestion par une file d'attente de la base des clauses ("FIFO") dans la section 4. Dans la section 5, nous présentons notre première stratégie de réduction appelée stratégie aléatoire bornée par la taille ("Size-Bounded Randomized strategy"). Dans la section 6, nous décrivons plusieurs stratégies de suppression qui nous permettent de garder les clauses apprises qui sont plus susceptibles de couper des branches en haut de l'arbre de recherche. Toutes ces stratégies sont intégrées dans le solveur Minisat 2.2 et comparées aux solveurs SAT de l'état de l'art *Glucose* 3.0 et *Lingeling* 2013. Nous présentons également les résultats obtenus en intégrant les stratégies basées sur la taille bornée dans *Glucose* 3.0. Avant de conclure, une discussion est fournie dans la section 7.

2 Travaux Connexes

Dans cette section, nous décrivons certaines stratégies de nettoyage de la base des clauses apprises telles que décrites par les auteurs. A notre connaissance, la stratégie visant à maintenir un ensemble pertinent de clauses remonte aux développements de procédures de preuves efficaces basées sur la résolution en démonstration automatique. Dans [17], D. Gelperin propose différentes stratégies qui tentent de determiner la satisfiabilité d'un ensemble de clauses tout en minimisant en même temps la taille de l'ensemble des clauses retenues. Dans les problèmes de satisfaction de contraintes et de satisfiabilité propositionnelle, pour surmonter le surcoût d'apprentissage sans restriction, plusieurs stratégies ont été proposées par Bayardo et al. [8, 7], y compris celles d'apprentissages par taille bornée ("Bounded Learning") et par relevance ("Relevance Based Learning"). Ils ont défini la taille bornée (respectivement, la relevance) d'ordre i, qui conserve indéfiniment uniquement les clauses raisons dérivées contenant au plus i variables (respectivement, maintient toute clause raison dérivée qui contient au plus i variables dont les affectations ont changé depuis que la raison a été dérivée). Cette question a d'abord été abordée dans GRASP par Joao Marques Silva et al. [28]. En effet, afin de garantir que la croissance de la base des clauses soit polynomiale en nombre de variables dans le pire des cas, dans [28] les auteurs proposent une stratégie selective sur les clauses qui doivent être ajoutées à la base de clauses. Plus précisément, étant donné un paramètre entier k, les clauses conflits dont la taille (nombre de littéraux) est inférieure à k sont marquées en vert (ajoutées à la base des clauses) tandis que celles de taille supérieure à k sont marquées en rouge et conservées autour uniquement lorsqu'elles sont des clauses unitaires, c'est-à-dire, une clause rouge est supprimée lorsqu'elle contient plus d'un littéral libre (non assigné).

Comme beaucoup d'autres solveurs, Chaff [30] intègre la suppression des clauses conflits ajoutées pour éviter une explosion de la mémoire. Chaff utilise essentiellement les stratégies de suppression de clauses paresseuses planifiées. Lorsqu'une clause est ajoutée, elle est analysée afin de déterminer à quel moment dans l'avenir, le cas échéant, la clause doit être supprimée. La mesure utilisée est la pertinence, de telle sorte que lorsque plus de n (où n est typiquement 100-200) littéraux dans la clause deviendront non-assignés pour la première fois, la clause sera marquée comme étant supprimée. La mémoire réelle associée aux clauses supprimées est récupérée régulièrement avec une étape de compactage.

Dans Berkmin [19], les auteurs considèrent que les clauses les plus récemment déduites sont plus précieuses parce qu'il a fallu plus de temps pour les déduire de l'ensemble des clauses originales. La base des clauses est considérée comme une file d'attente ("First In First Out"). La stratégie de suppression de Berkmin maintient les clauses de petites tailles (de taille inférieure à 8) combinée avec la représentation en file d'attente de la base des clauses apprises.

Minisat [16] supprime agressivement les clauses apprises sur la base d'une activité heuristique similaire à celle des variables. Une clause apprise est considérée comme non pertinente si son activité ou sa participation à l'analyse de conflits récente est marginale [16]. La limite sur le nombre de clauses apprises autorisées augmente après chaque redémarrage. Cette stratégie a été améliorée dans MiniSAT 2.2.

Plus récemment, Audemard and Simon [5] utilise le nombre de niveaux différents ("LBD - Literal Block Distance") impliqués dans une clause apprise pour quantifier la qualité des clauses apprises. Cette measure a été identifiée et utilisée dans [3] pour prouver l'optimalité du schéma First UIP. Dans [5] les clauses ayant une plus petite valeur de LBD sont considérées comme étant plus pertinentes. La mesure LBD est intégrée au solveur MiniSAT conduisant au solveur Glucose, l'un des solveurs SAT de l'état de l'art. La mesure LBD est également exploitée dans les solveurs Lingeling [11], SAT13 [27] le solveur basé sur CDCL conçu par D. Knuth, et quelques autres solveurs SAT intro-

duits dans la dernière compétition SAT 2013 (exemples *gluebit_clasp* 1.0, *BreakIDGlucose* 1, *glueminisat* 2.2.7j). Une autre stratégie de gestion dynamique de la base des clauses apprises est proposée dans [4]. Elle est basée sur un principe de gel et d'activation dynamique des clauses apprises. À un état de la recherche donné, en utilisant une fonction de sélection pertinente basée sur une sauvegarde progressive des affectations des variables [32], elle active les clauses apprises les plus prometteuses tout en gelant celles jugées non pertinentes.

Dans [21], les auteurs proposent deux mesures pour prédire la qualité des clauses apprises. La première mesure est basée sur le niveau de retour-arrière (BTL), tandis que la seconde est basée sur une notion de distance, définie comme la différence entre le maximum et le minimum des niveaux d'affectations des littéraux impliqués dans la clause apprise.

Enfin, les stratégies d'échange de clauses par taille bornée sont également considérées dans plusieurs solveurs SAT parallèles basés sur le portfolio et diviser pour régner (par exemple ManySAT [22] et PMSat [18]).

3 Définitions et notations préliminaires

Une *formule CNF* $\mathcal{F}$ est une conjonction de *clauses*, où une clause est une disjonction de *littéraux*. Un littéral est interprété comme une variable propositionnelle positive (x) ou négative ($\neg x$). Les deux littéraux x et $\neg x$ sont appélés *complémentaire*. Soit c une clause, $|c|$ denote la taille de la clause c (son nombre de littéraux).

Une clause *unitaire* est une clause contenant seulement un seul littéral (appelé *littéral unitaire*), tandis qu'une clause binaire contient exactement deux littéraux. Une *clause vide* , notée $\bot$, est interprétée comme fausse (insatisfiable), alors qu'une *formule CNF vide*, notée $\top$, est interprétée comme vraie (satisfiable). L'ensemble des variables apparaissant dans $\mathcal{F}$ est noté $V_{\mathcal{F}}$. Un ensemble de littéraux est *complet* s'il contient un littéral pour chaque variable de $V_{\mathcal{F}}$, et *fondamental* s'il ne contient pas de littéraux complémentaires. Une *affectation* ρ d'une formule booléenne $\mathcal{F}$ est une fonction qui associe la valeur $\rho(x) \in \{false, true\}$ à quelques variables de $x \in \mathcal{F}$. ρ est dite *complète* si elle attribue une valeur pour chaque variable $x \in \mathcal{F}$, et *partielle* sinon. Une affectation est alternativement représentée par un ensemble de littéraux, i.e., $\rho = \bigcup_{x \in V_{\mathcal{F}}} f(x)$, où $f(x) = x$ (respectivement $f(x) = \neg x$), si $\rho(x) = vrai$ (respectivement $\rho(x) = faux$). Un *modèle* d'une formule $\mathcal{F}$ est une affectation ρ qui satisfait la formule.

Nous introduisons à présent quelques notations et terminologies sur les solveurs SAT basés sur la procédure de Davis Logemann Loveland, communément appelée DPLL [13]. DPLL [13] est une procédure de recherche de type *backtrack*; À chaque nœud les littéraux affectés (le littéral de décision et les littéraux propagés) sont étiquetés avec le même *niveau de décision*, initialisé à 1 et incrémenté à chaque nouveau point de décision. Le niveau de décision courant est le niveau le plus élevé dans la pile de propagation. Lors d'un retour-arrière ("*backtrack*"), les variables ayant un niveau supérieur au niveau du backtrack sont défaites et le niveau de décision courant est décrémenté en conséquence (égal au niveau du backtrack). Au niveau i, l'interprétation partielle courante ρ peut être représentée comme une séquence décision-propagations de la forme $\langle (x_k^i), x_{k_1}^i, x_{k_2}^i, \ldots, x_{k_{n_k}}^i \rangle$ où le premier littéral x_k^i correspond au littéral de décision x_k affecté au niveau i et chaque $x_{k_j}^i$ de l'ensemble $1 \leqslant j \leqslant n_k$ représente les littéraux unitaires propagés à ce même niveau i. Une telle interprétation partielle (séquence décision-propagations) associée à un nœud donné de l'arbre de recherche est appelée *interprétation partielle ordonnée*. Soit $x \in \rho$, on note $lev(x, \rho)$ le niveau d'affectation de x dans ρ. Soit ρ une interprétation partielle, et c une clause, nous définissons c^i comme la projection de c sur les littéraux de c affectés au niveau i, c'est à dire $c^i = \{x | lev(x, \rho) = i\}$. Cet ensemble est appelé bloc dans [5].

Un solveur SAT basé sur l'apprentissage de clause par analyse de conflit (CDCL) explore l'espace de recherche en effectuant successivement une séquence de décision/propagations. Quand un conflit est rencontré, une clause conflit est dérivée par résolution sur les clauses impliquées dans le processus de propagation unitaire (codé comme un graphe d'implications). Une telle clause conflit apprise est alors ajoutée à la base des clauses apprises. Elle nous permet de produire un littéral unitaire (assertif) à un niveau plus haut de l'arbre de recherche. Ensuite, le solveur SAT basé sur l'architecture CDCL effectue un retour-arrière non chronologique à ce niveau, propage le littéral assertif et répète la séquence de décison/propagations jusqu'à ce qu'un modèle soit trouvé ou alors une clause vide est dérivée. La recherche redémarre régulièrement, et la base des clauses apprises est régulièrement réduite par l'élimination des clauses non pertinentes. Ces différentes composantes sont liées entre elles. Par exemple le redémarrage a des effets très forts sur la composante d'apprentissage [23, 12, 34, 33, 22, 6]. Certains solveurs SAT de l'état de l'art comme *Glucose* utilisent un redémarrage agressif, très utile pour la résolution des instances insatisfiables.

4 Quelques Stratégies de Suppression des Clauses

Comme mentionné dans l'introduction, notre principal objectif est de quantifier tout d'abord l'écart de performance entre les stratégies de suppression des clauses apprises de l'état de l'art et quelques stratégies basiques, y compris celles basées sur un choix aléatoire des clauses à supprimer. Dans cette section, nous illustrons les performances des stratégies de suppression basées sur la taille, sur une suppression aléatoire ("Random") et sur une gestion en file d'attente de la base des clauses apprises ("FIFO"). Toutes ces stratégies statiques sont intégrées sans aucune autre modification à MiniSAT 2.2. A chaque nettoyage de la base des clauses apprises, nous maintenons uniquement la moitié des clauses de la base estimées comme plus importantes, l'autre moitié considérée comme moins importante est supprimée. Les trois nouvelles versions de MiniSAT sont obtenues comme suit :

- *Size-MiniSAT* : A chaque conflit, l'activité de la clause apprise c est égale à sa taille, c'est à dire $\mathcal{A}(c) = |c|$
- *Rand-MiniSAT* : A chaque conflit, l'activité de la clause apprise c est fixée à une value réelle aléatoire $w \in [0..1]$, c'est à dire $\mathcal{A}(c) = drand(random\_seed)$. Nous avons utilisé exactement la fonction aléatoire de MiniSAT avec le même paramètre random_seed pour obtenir des résultats reproductibles.
- *FIFO-MiniSAT* : Dans cette version, les clauses apprises sont gérées en utilisant une file d'attente. Chaque fois qu'une réduction est effectuée, les clauses les plus anciennes sont supprimées.

Solveurs	#Résolues(#SAT-#UNSAT)	Temps moyen
MiniSAT	201 (122 - 79)	956.78s
*Glucose*3.0	216 (104 - 112)	**807.62s**
*Lingeling*2013	**233** (119 - **114)**	1090.99s
Size-MiniSAT	220 (**126** - 94)	1226.27s
Rand-MiniSAT	191 (121 - 70)	1071.05s
FIFO-MiniSAT	173 (119 - 54)	862.50s

TABLE 1 – Une évaluation comparative des stratégies basées sur Taille/RANDOM/FIFO-MiniSAT

Pour toutes les stratégies de suppression définies dans ce chapitre, les clauses avec les plus petites activités sont considérées comme plus pertinentes. Concernant la fréquence de nettoyage, nous suivons la même stratégie mise en oeuvre dans MiniSAT 2.2. Pour toutes les expérimentations présentées dans ce chapitre, nous exécutons les solveurs SAT sur les 300 instances prises de la compétition SAT 2013. Toutes les instances sont pré-traitées par SatElite [15] avant l'exécution du solveur SAT. Nous utilisons une machine Quad-Core Intel XEON avec 32GB de mémoire fonctionnant à 2.66 Ghz. Pour chaque instance, nous utilisons un temps limite égal à 5000 secondes du temps CPU.

Dans la table 1, nous donnons l'évaluation expérimentale comparative de nos trois versions de *MiniSAT* avec *Glucose* 3.0 et *Lingeling* 2013 le meilleur solveur de la compétition SAT 2013 (sur la catégorie applications). Dans la deuxième colonne, nous donnons le nombre total d'instances résolues (#Résolues). Nous mentionnons également, le nombre d'instances prouvées satisfiables (#SAT) et insatisfiables (#UNSAT) entre parenthèses. La troisième colonne indique le temps CPU moyen en secondes (le temps total sur les instances résolues divisé par le nombre d'instances résolues).

A partir de cette première expérimentation, les solveurs SAT de l'état-de-l'art *Lingeling* 2013 et *Glucose* 3.0 résolvent 233 et 216 instances respectivement. Le solveur *MiniSAT* résout 201 instances. Essayons de commenter les performances des autres versions de *MiniSAT* sans aucune autre amélioration. Le solveur *Size-MiniSAT* est capable de résoudre 221 instances (5 instances de plus que *Glucose* 3.0). Mais *Glucose* 3.0 est meilleur en terme de temps CPU moyen. D'autres observations importantes peuvent être tirées de cette première expérimentation. Sur les instances satisfiables *Size-MiniSAT* est plus performant que tous les autres solveurs, il résout 126 instances. La dernière remarque que l'on peut tirer est le fait que *Rand-MiniSAT* résout plus d'instances satisfiables (121) que les solveurs de l'état-de-l'art *Glucose* 3.0 et *Lingeling* 2013. Enfin *FIFO-MiniSAT* est clairement le plus mauvais des solveurs, particulièrement sur les instances insatisfiables (seulement 54 instances sont résolues). Cependant, il reste compétitif sur les instances satisfiables et résout 15 instances de plus que *Glucose*

5 Stratégie Randomisée Bornée par la Taille

Les résultats remarquables obtenus dans la section 4 par *Size-MiniSAT*, où les clauses de petites tailles sont considérées comme étant plus pertinentes sont clairement encourageants. Dans cette section, nous proposons une nouvelle stratégie simple de réduction, appelée "Size-Bounded Randomized strategy" (en court SBR), qui maintient les clauses de petites tailles (de taille inférieure à k), tout en supprimant aléatoirement les clauses de taille plus grande que k. L'un

des principaux inconvénients de l'activité basée sur la taille est que les clauses de grandes tailles sont souvent considérées comme non pertinentes. Sur certains problèmes difficiles, une telle sélection drastique des clauses de petite taille pourrait avoir des effets négatifs. Pour pallier ce problème, nous avons introduit une certaine randomisation à l'approche basée sur la taille. Plus précisément, la stratégie SBR est définie comme suit : étant donné une borne supérieure k sur la longueur de la clause apprise, chaque fois qu'une clause apprise c est dérivée, si sa taille est plus petite ou égale à k alors $\mathcal{A}(c) = |c|$, sinon $\mathcal{A}(c) = k + drand(random\_seed)$. En d'autres termes nous préférons toujours les clauses courtes, tandis que les clauses de taille plus grande que k sont considérées comme ayant la même taille k avec une valeur aléatoire supplémentaire. De cette manière, les clauses de grandes tailles peuvent être sélectionnées de manière aléatoire. Le solveur $SBR(k)$-$MiniSAT$ (où k est la borne supérieure de la taille) implémente une telle stratégie. Pour déterminer la meilleure borne supérieure de la taille k, nous exécutons $SBR(k)$-$MiniSAT$ avec $k = 2$, 5, 10, 12 et 15.

Solveurs	#Résolues (#SAT - #UNSAT)	Temps moyen
Glucose 3.0	216 (104 - 112)	**807.62s**
Lingeling 2013	233 (119 - **114**)	1090.99s
$SBR(2)$-$MiniSAT$	196 (122 - 74)	1064.17s
$SBR(5)$-$MiniSAT$	218 (118 - 100)	1213.36s
$SBR(10)$-$MiniSAT$	231 (124 - 107)	1158.25s
$SBR(12)$-$MiniSAT$	**239** (**129** - 110)	1265.96s
$SBR(15)$-$MiniSAT$	228 (120 - 108)	1226.96s

TABLE 2 – Une évaluation comparative de $SBR(k)$-$MiniSAT$

Les résultats sont présentés dans la table 2. $SBR(12)$-$MiniSAT$ obtient les meilleures performances. Elle résout 239 instances, 6 instances de plus que *Lingeling* 2013 et 23 instances de plus que *Glucose* 3.0. Ces résultats remarquables remettent au goût du jour les anciennes stratégies basées sur la taille bornée, proposées il y a plus d'une quinzaine d'années [29, 7, 8]. Les résultats montrent également que l'ajout d'une certaine randomisation pour permettre la selection des clauses de grande taille nous permet d'introduire une certaine diversification à la dérivation par résolution des solveurs SAT de type CDCL. Il est important de noter que notre implémentation proposée peut être obtenue en ajoutant trois lignes de codes à $MiniSAT$ 2.2 sans aucun réglage ou améliorations supplémentaires. Il convient de signaler que les solveurs $SBR(k)$-$MiniSAT$, pour $k = 5, 10, 12, 15$, sont plus performants que le solveur *Glucose* 3.0.

6 Vers des Stratégies de Suppressions Basées sur la Pertinence

Renforcé par l'intérêt de garder les clauses de petites tailles, notre but dans cette section est de montrer que d'autres variantes de ces stratégies basées sur la taille peuvent conduire à une meilleure performance. Plus précisément, notre objectif est de concevoir des stratégies de suppressions dynamiques, où les activités des clauses apprises sont mises à jour au cours de la recherche. Dans la plupart des solveurs SAT de l'état de l'art, les activités des clauses apprises sont mises à jour dynamiquement.

6.1 Mesures Dynamiques de Pertinence des Clauses

Expliquons l'idée qui nous a guidée dans la conception de ces nouvelles variantes dynamiques. Intuitivement, *une clause apprise de petite taille avec les littéraux assignés le plus souvent en haut de l'arbre de recherche courant est considérée comme étant plus pertinente.*

Étant donné une interprétation partielle ρ et c une clause de la base des clauses apprises, notre première stratégie dynamique de suppression basée sur la taille est définie comme suit : L'activité initiale de c est égale à $|c|$. Supposons maintenant que $l \in \rho$ est assigné par propagation unitaire au niveau d, grâce à la clause apprise c. Dans ce cas la nouvelle activité de c est égale à d si $d < |c|$. Cette mesure nous permet de garder les clauses apprises qui sont plus susceptibles de couper les branches au sommet de l'arbre de recherche. Par exemple, pour deux clauses de taille égale, nous préférons celles contenant les littéraux assignés au sommet de l'arbre de recherche. En intégrant cette mesure dans MiniSAT 2.2, nous obtenons notre première variante nommée "$SizeD$-$MiniSAT$".

Suivant la même idée, nous proposons une seconde version nommée "$Size(k)D$-$MiniSAT$". Le but est d'introduire un seuil k sur la longueur des clauses, afin de mettre à jour uniquement l'activité des clauses ayant une activité supérieure à k, et en gardant statique l'activité des clauses courtes (clauses dont la taille est inférieure à k). Plus précisément, l'activité initiale de c est définie comme suit : si $|c| \leqslant k$ alors $\mathcal{A}(c) = |c|$, sinon $\mathcal{A}(c) = k + |c|$. Chaque fois qu'une clause apprise c est la raison d'un littéral propagé par propagation unitaire au niveau de décision d, son activité est mise à jour comme suit : si $k + d < \mathcal{A}(c)$ alors $\mathcal{A}(c) = k + d$. De cette façon, l'activité d'une clause de taille inférieure à k n'est pas mise à jour.

La troisième variante est définie comme suit : Soit ρ une interprétation partielle conduisant à un conflit au niveau de décision d et c sa clause apprise asso-

ciée. L'activité de c est initialement fixée à $\mathcal{A}(c) = \sum_{i=1}^{d} i \times |c^i|$. Au cours de la recherche, chaque fois que c est la raison de la propagation d'un littéral, et si sa nouvelle activité par rapport à l'interprétation courante est meilleure, l'activité de c est mise à jour. Une clause c est considérée meilleure qu'une clause c' si $\mathcal{A}(c) < \mathcal{A}(c')$. Cette nouvelle stratégie nous conduit à un nouveau solveur nommé "*RelD-MiniSAT*".

Solveurs	#Résolues(#SAT-#UNSAT)	Temps moyen
Glucose 3.0	216 (104 - 112)	**807.62s**
Lingeling 2013	**233** (119 - **114**)	1090.99s
Size(12)*D-MiniSAT*	**233** (**123** - 110)	1153.9s
SizeD-MiniSAT	231 (120 - 111)	1174.97s
RelD-MiniSAT	222 (122 - 100)	1157.31s

TABLE 3 – Une évaluation comparative des stratégies de suppressions dynamiques.

Dans la table 3, nous présentons les résultats comparatifs des trois nouvelles stratégies de suppressions dynamiques basées sur la pertinence. Comme nous pouvons voir, *Lingeling* 2013 et *Size*(12)*D-MiniSAT* présentent des performances comparables (233 instances résolues). Nous observons aussi que *Lingeling* 2013 est meilleur (respectivement mauvais) que *Size*(12)*D-MiniSAT* sur les instances insatisfiables (respectivement satisfiables). Le solveur MiniSAT intégrant nos trois stratégies de suppression basées sur la pertinence est plus performant que le solveur *Glucose* 3.0.

Dans cette section, trois stratégies dynamiques sont proposées. Dans ces stratégies, les activités des clauses apprises sont mises à jour dynamiquement durant la recherche. En résumé, la stratégie randomisée bornée par la taille *SBR*(12)-*MiniSAT* demeure la meilleure, puisqu'elle résout plus d'instances (239 instances résolues) que toutes les stratégies proposées dans ce travail, y compris celles des solveurs SAT de l'état-de-l'art *Lingeling* 2013 et *Glucose* 3.0.

Toutes les stratégies proposées peuvent être facilement intégrées à MiniSAT en utilisant quelques lignes de codes.

6.2 Substituer le LBD avec la Taille de la Clause dans Glucose

Dans les sections précédentes, nous avons comparé nos stratégies de suppressions intégrées à MiniSAT avec les solveurs SAT de l'état de l'art tels que *Glucose* 3.0 et *Lingeling* 2013. Ces deux solveurs exploitent la mesure basée sur le LBD pour la gestion de la base des clauses apprises. La question à laquelle nous souhaitons répondre est la suivante : qu'en est-il de la substitution de la mesure LBD par la taille de la clause dans *Glucose* 3.0 et *Lingeling* 2013 ? Comme ces deux solveurs exploitent les mesures basées sur le LBD pour la gestion de la base des clauses apprises, nous ne modifions que *Glucose* 3.0 pour évaluer l'effet de la substitution du LBD avec la taille dynamique des clauses ("*Size*(k)*D*") présentée dans la section 6.1. Le solveur SAT obtenu est appelé "*Size*(k)*D-Glucose* 3.0", où k est un *seuil* sur la taille de la clause. Ce nouveau solveur est obtenu en substituant la valeur du LBD par la taille de la clause dans les deux parties suivantes de *Glucose* 3.0 :

— *Activité initiale de la clause apprise* : Quand une clause apprise c est dérivée, son activité est initialisée comme suit : $\mathcal{A}(c) = |c|$ si $|c| \leqslant k$, sinon $\mathcal{A}(c) = k + |c|$.

— *Mise-à-jour dynamique de l'activité de la clause* : à chaque conflit, l'activité de chacune des clauses apprises impliquées dans la dérivation de la clause assertive est mise-à-jour comme suit : soit d le niveau de conflit. Si $k + d < \mathcal{A}(c)$ alors $\mathcal{A}(c) = k + d$. Évidemment, l'activité d'une clause de taille inférieure à k n'est pas mise-à-jour.

Le solveur *SBR*(k)*-Glucose* 3.0 intègre la stratégie de suppression de taille bornée randomisée. Il est obtenu comme suit : chaque fois qu'une clause apprise c est dérivée, elle reçoit $\mathcal{A}(c) = |c|$ si $|c| \leqslant k$, sinon $\mathcal{A}(c) = k + irand(random\_seed, |V_{\mathcal{F}}|)$, où $irand(random\_seed, |V_{\mathcal{F}}|)$ retourne un nombre entre 0 et $|V_{\mathcal{F}}|$. Cette activité reste inchangée durant la recherche.

Solveurs	#Résolues (#SAT - #UNSAT)	Temps moyen
Glucose 3.0	216 (104 - 112)	**807.62s**
Size(12)*D-Glucose*	**224** (103 - **121**)	919.16s
Size(15)*D-Glucose*	219 (100 - 119)	944.36s
SBR(15)*-Glucose*	222 (**105** - 117)	1001.34s

TABLE 4 – Une évaluation comparative de *Glucose*, *Size*(k)*D-Glucose* et *SBR*(k)*-Glucose*.

Cette expérimentation démontre que la taille de la clause est clairement plus pertinente que la mesure LBD/glue. Comme nous pouvons observer, à partir du résultat présenté dans la table 4, *Size*(12)*D-Glucose* 3.0 résout 8 instances de plus que *Glucose* 3.0. Plus intéressant encore, sur les instances insatisfiables, notre version résout 9 instances de plus que *Glucose* 3.0. Sur les instances satisfiables, les deux solveurs présentent un comportement similaire. *Glucose* 3.0 résout seulement une instance de plus que *Size*(12)*D-Glucose* 3.0. Cette expérimentation nous donne une illustration que la mesure basée sur la taille de la clause est meilleure que celle basée sur le LBD.

Instance	(#Var, #Cl)	Statut	Temps
ctl_4291_567_11_unsat	(17850, 147308)	N	4431.87
ctl_4291_567_11_unsat_pre	(15232, 142785)	N	4253.48
ctl_4291_567_8_unsat	(17850, 147312)	N	3562.12
ctl_4291_567_8_unsat_pre	(15232, 142789)	N	3407.92
nossum–sha1/22-160/002	(4128, 126580)	Y	2897.44
nossum–sha1/23-96/003	(4288, 132608)	Y	3005.76

TABLE 5 – $SBR(12)$-$MiniSAT$ sur les instances non résolues (Catégorie Application - Compétition SAT 2013).

6.3 Instances non résolues à la dernière compétition SAT 2013

Dans la table 5, nous présentons les résultats obtenus par $SBR(12)$-$MiniSAT$ sur les instances ouvertes prises de la compétition SAT 2013 (catégorie application). Pour chaque instance, nous reportons le nom de l'instance ($Instance$), son nombre de variables et de clauses (#Var, #Cl), le temps cpu (en secondes) utilisé par $SBR(12)$-$MiniSAT$ pour résoudre l'instance ($Temps$).

A la dernière compétition SAT 2013 [1], 12 instances de la catégorie application n'ont été résolues par aucun solveur séquentiel en moins de 5000 secondes. Parmi ces instances, 6 sont résolues par notre solveur $SBR(12)$-$MiniSAT$. Les résultats sont présentés dans la Table 5. Ces résultats confirment clairement les bonnes performances de notre solveur $SBR(12)$-$MiniSAT$.

7 Discussion

En résumé, nous démontrons que la taille de la clause reste la meilleure mesure pour quantifier l'utilité d'une clause donnée. La leçon qu'on peut tirer de cette étude est que la prédiction des meilleures clauses à maintenir durant la recherche mérite une enquête plus poussée. A notre avis, la performance de la mesure LBD peut être expliquée par le fait qu'elle est réellement liée à la taille des clauses. En effet, nous avons $2 \leqslant LBD(c) \leqslant |c|$. La taille de la clause est une borne supérieure de la mesure LBD. Par exemple, le LBD des clauses binaires est égal à 2. Par conséquent, la stratégie définie dans *Glucose* favorise dans un certain sens le maintien des clauses de petites tailles. Dans [5], les auteurs mentionnent que : "*Un bon schéma d'apprentissage devrait ajouter des liens explicites entre des blocs indépendants des littéraux propagés (ou de décision). Si le solveur reste dans le même espace de recherche, une telle clause aidera probablement à réduire le nombre de prochains niveaux de décision au cours du reste de la recherche*". L'intuition donnée par les auteurs pour comprendre l'importance des clauses de LBD égal à 2 est la suivante : "*la variable du dernier niveau de décision sera collée ("glued") avec le bloc de littéraux propagés à un niveau plus haut de l'arbre de recherche. Nous suspectons toutes ces clauses d'être vraiment importantes au cours de la recherche*". Essayons de discuter sur ces affirmations. Premièrement, toute clause apprise ajoute des liens explicites entre les blocs indépendants de littéraux propagés. Deuxièmement, toute clause apprise aide aussi à élaguer l'espace de recherche et donc à réduire le nombre de décisions. La question qu'on se pose est la suivante : soit $c_1 = (x_1 \vee x_2 \vee x_3)$ et $c_2 = (y_1 \vee y_2 \vee, \ldots, \vee y_n)$ avec $n > 3$, $LBD(c_1) = 3$ et $LBD(c_2) = 2$, laquelle est pertinente ? Comme vous pouvez le deviner, notre préférence va à la première clause.

Du côté théorique, la prédiction de la pertinence d'une clause donnée pour la preuve est liée au problème de trouver les plus courtes réfutations par résolution. Ce dernier problème a été démontré NP-Difficile [24], et aussi NP-Difficile en l'approximant avec un facteur constant [1]. Une deuxième mesure importante est la largeur d'une preuve par résolution, définie comme étant le nombre maximal de littéraux dans une clause de la preuve. Dans [10], une relation intéressante est établie entre la largeur de la preuve et la longueur de la preuve. Ces résultats suggèrent que la taille des clauses apprises est un concept fondamental pour la résolution pratique du problème SAT. Sur certaines instances difficiles, on a besoin de maintenir les clauses de grande taille. Notre stratégie aléatoire est inspirée de ces résultats.

Une remarque importante, toutes les stratégies présentées dans ce papier peuvent être intégrées facilement (quelques lignes de code) à tout solveur basé sur l'architecture CDCL. Une page web dédiée, incluant les stratégies proposées pour la réduction de la base des clauses apprises intégrées à MiniSAT est actuellement en construction (http ://www.cril.fr/∼sais).

8 Conclusion et Perspectives

Dans ce papier, nous avons abordé le problème de gestion de la base des clauses apprises. Nous avons démontré que les stratégies d'apprentissage basées sur la taille bornée proposées il y a plus d'une quinzaine d'années [29, 7, 8] restent de bonnes mesures pour prédire la qualité des clauses apprises. Nous avons également montré que l'ajout de la randomisation à l'apprentissage en fixant une borne supérieure sur la taille est un bon moyen de parvenir à une diversification contrôlée. Cette stratégie nous permet de privilégier les clauses de petite taille, tout en maintenant une petite fraction de clauses de grande taille néces-

1. instances non résolues *http ://edacc4.informatik.uni-ulm.de/SC*13/*experiment*/19/*unsolved-instances*/

saires pour la dérivation des preuves par résolution sur certaines instances. Les résultats expérimentaux, montrent que la stratégie basée sur la taille bornée randomisée intégrée à MiniSAT peut atteindre des performances meilleures que celles des solveurs SAT de l'état de l'art. Convaincu par l'importance des clauses de petites tailles, nous avons proposé plusieurs variantes dynamiques efficaces qui visent à maintenir des clauses courtes contenant des littéraux qui apparaissent le plus souvent en haut de l'arbre de recherche. Notre dernière évaluation montre que la substitution de la mesure LBD par la mesure basée sur la taille dans *Glucose* 3.0 mène à une meilleure performance à la fois sur les instances satisfiables et sur les instances insatisfiables.

Références

[1] Michael Alekhnovich, Sam Buss, Shlomo Moran, and Toniann Pitassi. Minimum propositional proof length is np-hard to linearly approximate. In Lubos Brim, Jozef Gruska, and Jiri Zlatuska, editors, *Mathematical Foundations of Computer Science 1998*, volume 1450 of *Lecture Notes in Computer Science*, pages 176–184. Springer Berlin Heidelberg, 1998.

[2] G. Audemard and L. Simon. Predicting learnt clauses quality in modern sat solvers. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence, IJCAI'09*, pages 399–404, 2009.

[3] Gilles Audemard, Lucas Bordeaux, Youssef Hamadi, Said Jabbour, and Lakhdar Sais. A generalized framework for conflict analysis. Technical Report MSR-TR-2008-34, Microsoft Research, Feb 2008.

[4] Gilles Audemard, Jean-Marie Lagniez, Bertrand Mazure, and Lakhdar Saïs. On freezing and reactivating learnt clauses. In *Fourteenth International Conference on Theory and Applications of Satisfiability Testing(SAT'11)*, jun 2011.

[5] Gilles Audemard and Laurent Simon. Predicting learnt clauses quality in modern sat solver. In *Twenty-first International Joint Conference on Artificial Intelligence(IJCAI'09)*, pages 399–404, jul 2009.

[6] Gilles Audemard and Laurent Simon. Refining restarts strategies for sat and unsat. In *18thInternational Conference on Principles and Practice of Constraint Programming(CP'2012)*, pages 118–126, 2012.

[7] Roberto J. Bayardo and Daniel P. Miranker. A complexity analysis of space-bounded learning algorithms for the constraint satisfaction problem. In *In Proceedings of the Thirteenth National Conference on Artificial Intelligence*, pages 298–304, 1996.

[8] Roberto J. Bayardo Jr. and Robert C. Schrag. Using csp look-back techniques to solve real-world sat instances. In *Proceedings of the Fourteenth American National Conference on Artificial Intelligence (AAAI'97)*, pages 203–208, Providence (Rhode Island, USA), July 1997.

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

[10] E. Ben-Sasson and A. Wigderson. Short proofs are narrow-resolution made simple. In *Computational Complexity, 1999. Proceedings. Fourteenth Annual IEEE Conference on*, pages 2–, 1999.

[11] A. Biere. Lingeling and friends entering the sat challenge 2012. In A. Balint, A. Belov, A. Diepold, S. Gerber, M. Jarvisalo, , and C. Sinz (editors), editors, *Proceedings of SAT Challenge 2012 : Solver and Benchmark Descriptions*, pages 33–34, University of Helsinki, 2012. vol. B-2012-2 of Department of Computer Science Series of Publications B.

[12] Armin Biere. Adaptive restart strategies for conflict driven sat solvers. In Hans Kleine Büning and Xishun Zhao, editors, *Theory and Applications of Satisfiability Testing - SAT 2008*, volume 4996 of *Lecture Notes in Computer Science*, pages 28–33. Springer Berlin / Heidelberg, 2008.

[13] M. Davis, G. Logemann, and D. W. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7) :394–397, 1962.

[14] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem proving. *Journal of the Association for Computing Machinery*, 5 :394–397, 1962.

[15] Niklas Eén and Armin Biere. Effective preprocessing in sat through variable and clause elimination. In *Proceedings of the Eighth International Conference on Theory and Applications of Satisfiability Testing (SAT'05)*, volume 3569 of *Lecture Notes in Computer Science*, pages 61–75, St. Andrews, Scottland, June 2005. Springer.

[16] Niklas Eén and Niklas Sörenson. An extensible sat-solver. In *Proceedings of the Sixth International Conference on Theory and Applications of Satisfiability Testing (SAT'03)*, volume 2929 of *Lecture Notes in Computer Science*, pages 333–336, Santa Margherita Ligure, Italy, May 2003. Published in 2004. Springer.

[17] David Gelperin. Deletion-directed search in resolution-based proof procedures. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, IJCAI'73, pages 47–50, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc.

[18] Luís Gil, Paulo Flores, and Luís M. Silveira. PMSat : a parallel version of MiniSAT. *Journal on Satisfiability, Boolean Modeling and Computation*, 6 :71–98, 2008.

[19] Eugene Goldberg and Yakov Novikov. Berkmin : A fast and robust sat-solver. *Discrete Applied Mathematics*, 155(12) :1549 – 1561, 2007.

[20] Carla P. Gomes, Bart Selman, and Henry Kautz. Boosting combinatorial search through randomization. In *Proceedings of the Fifteenth American National Conference on Artificial Intelligence (AAAI'97)*, pages 431–437, Madison, Wisconsin, USA, July 1998. American Association for Artificial Intelligence Press.

[21] Long Guo, Saïd Jabbour, and Lakhdar Saïs. Stratégies d'élimination des clauses apprises dans les solveurs sat modernes. In *9ièmes Journées Francophones de Programmation par Contraintes (JFPC'13)*, pages 147–156, Aix en provence, France, 2013.

[22] Youssef Hamadi, Said Jabbour, and Lakhdar Sais. ManySAT : a Parallel SAT Solver. *Journal on Satisfiability, Boolean Modeling and Computation*, 6 :245–262, 2009.

[23] Jinbo Huang. The effect of restarts on the efficiency of clause learning. In *Proceedings of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI'07)*, pages 2318–2323, Hyderabad, India, January 6-16 2007.

[24] Kazuo Iwama. Complexity of finding short resolution proofs. In Igor Privara and Peter Ruzicka, editors, *Mathematical Foundations of Computer Science 1997*, volume 1295 of *Lecture Notes in Computer Science*, pages 309–318. Springer Berlin Heidelberg, 1997.

[25] Hadi Katebi, KaremA. Sakallah, and Joao P. Marques-Silva. Empirical study of the anatomy of modern sat solvers. In KaremA. Sakallah and Laurent Simon, editors, *Theory and Applications of Satisfiability Testing - SAT 2011*, volume 6695 of *Lecture Notes in Computer Science*, pages 343–356, 2011.

[26] H. Kautz, E. Horvitz, Y. Ruan, C. Gomes, and B. Selman. Dynamic restart policies. In *aaai02*, pages 674–682, 2002.

[27] D. Knuth. Sat13. http ://www-cs-faculty.stanford.edu/∼uno/programs/sat13.w.

[28] João P. Marques-Silva and Karem A. Sakallah. Grasp—a new search algorithm for satisfiability. In *ICCAD '96 : Proceedings of the 1996 IEEE/ACM international conference on Computer-aided design*, pages 220–227, Washington, DC, USA, 1996. IEEE Computer Society.

[29] João P. Marques-Silva and Karem A. Sakallah. Grasp—a new search algorithm for satisfiability. In *ICCAD '96 : Proceedings of the 1996 IEEE/ACM international conference on Computer-aided design* [28], pages 220–227.

[30] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff : engineering an efficient sat solver. In *DAC '01 : Proceedings of the 38th annual Design Automation Conference*, pages 530–535, New York, NY, USA, June 2001. ACM.

[31] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff : Engineering an efficient sat solver. In *38th Design Automation Conference (DAC'01)*, pages 530–535, 2001.

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

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

[34] Knot Pipatsrisawat and Adnan Darwiche. Width-based restart policies for clause-learning satisfiability solvers. In *Proceedings of the 12th International Conference on Theory and Applications of Satisfiability Testing*, SAT '09, pages 341–355, Berlin, Heidelberg, 2009. Springer-Verlag.

[35] Knot Pipatsrisawat and Adnan Darwiche. On modern clause-learning satisfiability solvers. *Journal of Automated Reasoning*, 44(3) :277–301, 2010.

[36] João P. Marques Silva and Karem A. Sakallah. Grasp - a new search algorithm for satisfiability. In *International Conference on Computer-Aided Design (ICCAD'96)*, pages 220–227, 1996.

[37] Lintao Zhang, Conor F. Madigan, Matthew H. Moskewicz, and Sharad Malik. Efficient conflict driven learning in a boolean satisfiability solver. In *ICCAD '01 : Proceedings of the 2001 IEEE/ACM international conference on Computer-aided design*, pages 279–285, Piscataway, NJ, USA, November 2001. IEEE Press.

Détection de contraintes de cardinalité dans les CNF *

Armin Biere[1] **Daniel Le Berre**[2] **Emmanuel Lonca**[2] **Norbert Manthey**[3]

[1] Johannes Kepler University
[2] CNRS Université d'Artois
[3] Technische Universität Dresden
biere@jku.at {leberre,lonca}@cril.fr norbert.manthey@tu-dresden.de

Résumé

Dans cet article, nous présentons de nouvelles approches pour la détection de contraintes de cardinalité exprimées en CNF. La première est basée sur une analyse syntaxique des structures de données particulières des solveurs SAT utilisées pour représenter des clauses binaires et ternaires. La deuxième est basée sur une analyse sémantique à base de propagation unitaire. L'approche syntaxique calcule une approximation de l'ensemble des contraintes AtMost-1 et AtMost-2 de manière efficace alors que l'approche sémantique a l'avantage d'être générique, au sens où elle est capable de détecter des contraintes AtMost-k pour k quelconque, en un temps de calcul plus important. Nos expérimentations suggèrent que les deux approches sont efficaces en pratique pour extraire des contraintes AtMost-1 et AtMost-2.

Abstract

We present novel approaches to detect cardinality constraints expressed in CNF. The first approach is based on a syntactic analysis of specific data structures used in SAT solvers to represent binary and ternary clauses, whereas the second approach is based on a semantic analysis by unit propagation. The syntactic approach computes an approximation of the cardinality constraints AtMost-1 and AtMost-2 constraints very fast, whereas the semantic approach has the property to be generic, i.e. it can detect cardinality constraints AtMost-k for any k, at a higher computation cost. Our experimental results suggest that both approaches are efficient at recovering AtMost-1 and AtMost-2 cardinality constraints.

*Ce travail a été financé en partie par l'ANR TUPLES.

1 Introduction

À l'heure actuelle, les benchmarks au format CNF contiennent un grand nombre de fonctions booléennes encodées par des clauses [29, 15]. Parmi elles se trouvent les contraintes de cardinalité $\sum_{i=1}^{n} l_i \otimes k$ où $\otimes \in \{<, \leq, =, \geq, >\}$, des fonctions booléennes dont la satisfiabilité est déterminée en comptant le nombre de littéraux satisfaits dans le membre de gauche, et en le comparant avec le nombre du membre de droite, le *degré* de la contrainte. Par exemple, la contrainte $x_1 + x_2 + \neg x_3 + \neg x_4 \leq 2$ est satisfaite si et seulement si au moins deux de ses littéraux sont satisfaits. Un cas répandu d'utilisation de ces contraintes est d'exprimer qu'une variable de domaine de taille supérieure à 2 prenne exactement une valeur parmi un ensemble discret de valeurs. Puisque les contraintes de cardinalité sont des fonctions booléennes, il est possible de les exprimer *via* une CNF équivalente. L'approche "théorique", que l'on retrouve par exemple dans [12], traduit la contrainte $\sum_{i=1}^{n} l_i \leq k$ en utilisant $\binom{n}{k+1}$ clause négatives de taille $k+1$. Cet encodage est appelé *binomial encoding* en référence au nombre de clauses générées. En pratique, les encodages basés sur des ajouts de variables pour réduire le nombre de clauses améliorent généralement les temps de calcul. De multiples encodages ont été proposés ces dernières années (voir par exemple [14]). Nous étudions les encodages populaires dans la prochaines section. Les solveurs pseudo-booléens utilisent un système de preuves équivalent à la *résolution généralisée* [21], un cas particulier des *plans coupes* [12] qui *p-simule* la résolution. Cela implique par exemple que ces solveurs sont capables de résoudre des instances de pigeons [19] quand elles sont données sous forme de contraintes de cardinalité mais

pas quand elles sont exprimées sous forme de clauses. La raison de cet restriction est qu'appliquer la résolution généralisée sur des clauses est équivalent à la résolution [21], quand l'appliquer sur des contraintes de cardinalité revient à utiliser les plans coupes [12]. L'extraction de contraintes de cardinalité depuis des clauses en utilisant les plans coupes requiert une procédure spécifique. Considérons par exemple la contrainte $x_1 + x_2 + x_3 + x_4 \leq 1$, équivalente à $\neg x_1 + \neg x_2 + \neg x_3 + \neg x_4 \geq 3$. Cette contrainte peut être représentée en CNF *via* les clauses $\neg x_1 \vee \neg x_2, \neg x_1 \vee \neg x_3, \neg x_1 \vee \neg x_4, \neg x_2 \vee \neg x_3, \neg x_2 \vee \neg x_4$ et $\neg x_3 \vee \neg x_4$. Ces clauses peuvent aussi être représentées par les contraintes de cardinalité binaires $x_1 + x_2 \leq 1, x_1 + x_3 \leq 1, x_1 + x_4 \leq 1, x_2 + x_3 \leq 1, x_2 + x_4 \leq 1$ et $x_3 + x_4 \leq 1$. L'extraction de la contrainte de cardinalité originale depuis les clauses représentées par les contraintes de cardinalité binaires nécessite de dériver des contraintes intermédiaires ([12]) :

$$\frac{\begin{array}{c} x_1 + x_2 \leq 1 \\ x_1 + x_3 \leq 1 \\ x_2 + x_3 \leq 1 \end{array}}{x_1 + x_2 + x_3 \leq 1} \qquad \frac{\begin{array}{c} x_1 + x_2 \leq 1 \\ x_1 + x_4 \leq 1 \\ x_2 + x_4 \leq 1 \end{array}}{x_1 + x_2 + x_4 \leq 1}$$

$$\frac{\begin{array}{c} x_1 + x_3 \leq 1 \\ x_1 + x_4 \leq 1 \\ x_3 + x_4 \leq 1 \end{array}}{x_1 + x_3 + x_4 \leq 1} \qquad \frac{\begin{array}{c} x_2 + x_3 \leq 1 \\ x_2 + x_4 \leq 1 \\ x_3 + x_4 \leq 1 \end{array}}{x_2 + x_3 + x_4 \leq 1}$$

En ce qui concerne la première dérivation, la somme les trois contraintes de cardinalité nous donne $2x_1 + 2x_2 + 2x_3 \leq 3$, qui peut être réduite à $x_1 + x_2 + x_3 \leq 1$ en divisant l'inéquation par deux et en arrondissant le degré à l'entier inférieur. La même procédure est appliquée pour les trois autres dérivations. Finalement, la somme des quatre contraintes dérivées nous donne $3x_1 + 3x_2 + 3x_3 + 3x_4 \leq 4$. La contrainte attendue est obtenue après division par 3 et arrondi.

Le processus que nous venons de décrire est fastidieux et difficile à intégrer dans un solveur, d'où l'idée de tenter de détecter ces contraintes lors d'une phase de prétraitement, de manière indépendante du système de preuve utilisé par le solveur. Notre motivation quant à ce travail était de permettre aux solveurs de tirer avantage de ces contraintes de cardinalité, que ce soit en terme d'espace (gestion native), ou en terme de temps de calcul (meilleur système de preuve : résolution généralisée [21], plans coupes [12]). La détection des contraintes de cardinalité peut aussi être utile aux solveurs SAT purs, en leur permettant de réencoder des contraintes détectées en utilisant un encodage dont on peut espérer qu'il augmentera l'efficacité du solveur [26, 25]. Cela peut notamment être le cas quand l'encodage original était l'encodage binomial (souvent utilisé pour les contraintes AtMost-1).

2 Rappel succint d'encodages connus

Avant de traiter de la détection de contraintes de cardinalité, nous introduisons quelques-uns des encodages les plus populaires.

Pour une contrainte AtMost-1 $\sum_{i=1}^{n} x_i \leq 1$, l'idée derrière le *naïve encoding*, aussi appelé *pairwise encoding* ou *binomial encoding* (ou même *direct encoding* dans la communauté CP [33]), est d'exclure toute paire de littéraux satisfaits de manière explicite : $\bigwedge_{i=1}^{n} \bigwedge_{j>i}^{n} (\neg x_i \vee \neg x_j)$. De cette manière, l'encodage d'une contrainte portant sur n littéraux nécessite $\frac{n(n-1)}{2}$ clauses.

Le *nested encoding* utilise quant à lui des variables auxiliaires, dans le but de diminuer le nombre de clauses en séparant de manière récursive chaque contrainte en deux contraintes plus petites :

$$\textstyle\sum_{i=1}^{n} x_i \leq 1 = \\ (y + (\sum_{i=1}^{\lfloor \frac{n}{2} \rfloor - 1} x_i) \leq 1) \wedge (\neg y + (\sum_{i=\lfloor \frac{n}{2} \rfloor - 1}^{n} x_i) \leq 1).$$

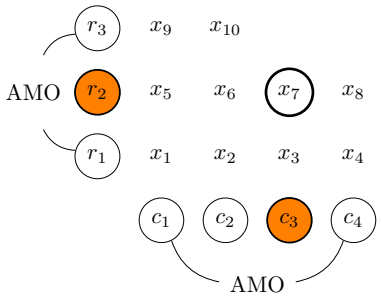
Pour $n = 4$, le *naïve encoding* nécessite 6 clauses, tout comme le nested encoding, qui nécessite en revanche plus de variables. Ainsi, on arrête de séparer les contraintes dès qu'elle contiennent au plus 4 littéraux. De cette manière, le nested encoding requiert $3n - 6$ clauses.

À l'heure actuelle, le meilleur encodage de contraintes AtMost-1 connu pour un grand nombre de variables (à partir de $n > 47$ [25]) est le *two product encoding* [11]. Pour n variables dans la contrainte, deux entiers $p = \lfloor \sqrt{n} \rfloor$ et $q = \lceil \frac{n}{p} \rceil$ sont utilisés comme indices de ligne et de colonne. Les variables x_i sont placées dans la matrice de telle sorte que chaque variable soit définie par un unique couple d'indices (r_s, c_t) grâce aux clauses $\neg x_i \vee r_s$ et $\neg x_i \vee c_t$, où $s = \lfloor \frac{i-1}{q} \rfloor + 1$ et $t = ((i-1) \mod q) + 1)$. Un exemple à 10 variables est donné à la figure 1.

Beaucoup d'autres encodages ont été proposés pour les contraintes AtMost-1 ([33, 18, 17, 3, 23, 14, 20, 7] et pour les contraintes AtMost-k avec $k > 1$ [1, 6, 30], certains utilisant des structures complexes [13, 5, 9]. Une étude récente approfondie l'efficacité pratique de ces encodages dans le contexte du problème MaxSAT [27].

3 Détection statique de contraintes AtMost-1 et AtMost-2

La détection du *naïve encoding* des contraintes AtMost-1 peut être réalisé par une analyse syntaxique de la formule, plus précisément en recherchant des cliques dans le *NAND graph* (NAG) de la formule (un graphe non-orienté où les littéraux liés sont les négations des littéraux des clauses binaires de la formules).



Clauses de sélection

$(\neg x_1 \vee r_1)$ $(\neg x_1 \vee c_1)$ $(\neg x_2 \vee r_1)$
$(\neg x_2 \vee c_2)$ $(\neg x_3 \vee r_1)$ $(\neg x_3 \vee c_3)$
$(\neg x_4 \vee r_1)$ $(\neg x_4 \vee c_4)$ $(\neg x_5 \vee r_2)$
$(\neg x_5 \vee c_1)$ $(\neg x_6 \vee r_2)$ $(\neg x_6 \vee c_2)$
$(\neg x_7 \vee r_2)$ $(\neg x_7 \vee c_3)$ $(\neg x_8 \vee r_2)$
$(\neg x_8 \vee c_4)$ $(\neg x_9 \vee r_3)$ $(\neg x_9 \vee c_1)$
$(\neg x_{10} \vee r_3)$ $(\neg x_{10} \vee c_2)$

AtMost-1 finales

$(\neg r_1 \vee \neg r_2)$ $(\neg r_1 \vee \neg r_3)$ $(\neg r_2 \vee \neg r_3)$
$(\neg c_1 \vee \neg c_2)$ $(\neg c_1 \vee \neg c_3)$ $(\neg c_1 \vee \neg c_4)$
$(\neg c_2 \vee \neg c_3)$ $(\neg c_2 \vee \neg c_4)$ $(\neg c_3 \vee \neg c_4)$

FIGURE 1 – Encodage de la contrainte AtMost-1 $\sum_{i=1}^{i\leq 10} x_i \leq 1$ avec le two product encoding, et deux contraintes AtMost-1 auxiliaires, $r_1 + r_2 + r_3 \leq 1$ et $c_1 + c_2 + c_3 + c_4 \leq 1$.

Dans [4, 2], les auteurs modifient les solveurs zChaff et Satz pour reconnaître ces contraintes en utilisant la propagation unitaire et la recherche locale. Une structure de données spécifique est souvent intégrée dans les solveurs SAT modernes pour réduire la taille en mémoire. Depuis un tel graphe, on peut extraire les contraintes AtMost-1 par une analyse syntaxique. La détection du *naïve encoding* des contraintes AtMost-2 peut être effectuée en s'intéressant aux clauses ternaires.

Les outils 3MCARD [32], LINGELING [10] et SBSAT [34] sont capables de reconnaître de telles contraintes en procédant à une analyse syntaxique ; nous présentons ci-dessous leurs procédures en plus de notre nouvelle méthode.

3MCARD et SBSAT ne restreignent pas leur recherche aux clauses d'une certaine taille, mais considèrent la formule dans son ensemble. SBSAT construit des BDDs en utilisant les clauses qui partagent des littéraux, et détecte les contraintes en fusionnant et analysant ces BDDs [34]. 3MCARD construit un graphe à partir des clauses binaires, et augmente la taille de la contrainte courante en collectant de nouvelles clauses. Seul Lingeling possède des méthodes spécialement conçues pour la détection des contraintes AtMost-1 et AtMost-2.

3.1 Détection du pairwise encoding

Détecter la structure du pairwise encoding dans un NAG est facile : si une clique est présente, alors les littéraux des nœuds correspondants forment une contrainte AtMost-1. Puisque la recherche d'une clique de taille k dans un graphe est NP-complet [22], une phase de prétraitement ne devrait pas effectuer une recherche exhaustive de cliques.

L'algorithme glouton de recherche de cliques implémenté dans LINGELING itère sur l'ensemble des littéraux n qui n'ont pas encore été inclus dans une contrainte AtMost-1. Pour chaque n, l'ensemble l'ensemble S de littéraux candidats est initialisé avec n. Ensuite, chaque littéral l qui apparaît nié dans une clause binaire $\neg n \vee \neg l$ (l et n sont connectés dans le graphe) est ajouté de manière gloutonne dans l'ensemble S après avoir vérifié que pour chaque littéral k ajouté précédemment dans S, l et k sont reliés dans le graphe. L'ensemble S finalement obtenu est une clique, et représente donc une contrainte AtMost-1 ($\sum_{l\in S} l \leq 1$), non-triviale dans le cas où $|S| > 2$ [10].

3.2 Détection du nested encoding

Considérons le *nested encoding* de la contrainte AtMost-1 $x_1 + x_2 + x_4 + x_5 \leq 1$, où la contrainte est divisée en $x_1 + x_2 + x_3 \leq 1$ et $\neg x_3 + x_4 + x_5 \leq 1$. Ces deux contraintes sont représentées en CNF par les 6 clauses $(\neg x_1 \vee \neg x_2), (\neg x_1 \vee \neg x_3), (\neg x_2 \vee \neg x_3), (x_3 \vee \neg x_4), (x_3 \vee \neg x_5), (\neg x_4 \vee \neg x_5)$. Puisque nous ne disposons pas de la contrainte binaire $(\neg x_1 \vee \neg x_4)$, la méthode présentée au-dessus ne peut être appliquée pour détecter cette contrainte. Nous présentons ici une nouvelle façon de détecter ces contraintes.

Les deux contraintes issues de la division de la contrainte originale peuvent être reconnues par la méthode de la section précédente (leurs littéraux forment une clique dans le NAG). De ce fait, on détecte qu'il existe une contrainte AtMost-1 qui porte sur le littéral x_3, et une autre qui porte sur $\neg x_3$. Par résolution, on obtient la contrainte recherchée.

Pour chaque variable v, toutes les paires de AtMost-1 (A,B) telles que v apparaît dans une contrainte et $\neg v$ apparaît dans l'autre sont sommées et simplifiées. L'étape de simplification vérifie qu'il n'y a pas de littéraux dupliqués (dans ce cas, les littéraux doivent être falsifiés puisque leur poids est supérieur au degré), ou de littéraux complémentaires (ici, tous les littéraux de la contrainte $(A + B)$ à l'exception des littéraux complémentaires doivent être falsifiés, puisque $x + \neg x = 1$

implique que le degré doit être réduit de 1, c'est-à-dire nul). La contrainte simplifiée est finalement ajoutée à S puis retournée par l'algorithme.

Puisque le *nested encoding* peut être encodé de manière récursive, l'algorithme peut être appelé de multiples fois pour trouver ces contraintes de manière récursive. Pour éviter de considérer plusieurs fois les mêmes couples de contraintes, on peut ne considérer que les couples pour lesquels un des deux éléments est une contrainte qui a été découverte à l'itération précédente.

3.3 Détection du two product encoding

Le *two product encoding* a une structure récursive similaire à celle du *nested encoding*; sa structure est cependant plus complexe. Nous détaillons donc cet algorithme plus en détail dans cette section. La figure 1 illustre une contrainte AtMost-1 encodée *via* le *two product encoding*.

Pour chaque littéral concerné, $x_1, ..., x_{10}$ dans notre exemple, deux implications sont utilisées pour fixer les indices de ligne et de colonne. Par exemple, puisque x_7 appartient à la deuxième ligne et la quatrième colonne, on ajoute les contraintes $x_7 \rightarrow r_2$ et $x_7 \rightarrow c_3$. Afin d'empêcher deux indices de ligne ou deux indices de colonne d'être sélectionnés simultanément, on ajoute des contraintes AtMost-1 sur les c_i et sur les r_i. Ces contraintes sont ajoutées avec le *pairwise encoding* si leur taille est faible, ou avec le *two product encoding* si leur taille est plus importante, d'où la nécessité d'une étape récursive dans l'algorithme.

Dans notre exemple, les contraintes de sélection d'indices pour x_7, $x_7 \rightarrow c_3$ et $x_7 \rightarrow r_2$ sont impliquées par le codage. De plus, les implications $c_3 \rightarrow \neg c_2$ et $\neg c_2 \rightarrow (\neg x_2 \wedge \neg x_6)$ montrent par transitivité que $x_7 \rightarrow (\neg x_6 \wedge \neg x_2)$. Puisque toutes les implications sont construites sur des clauses binaires, le raisonnement inverse tient aussi : $x_6 \rightarrow \neg x_7$ et $x_2 \rightarrow \neg x_7$; on peut alors déduire $x_6 + x_7 \leq 1$ et $x_2 + x_7 \leq 1$. Cependant, la contrainte $x_2 + x_6 \leq 1$ ne peut être déduite *via* les colonnes et leurs littéraux c_2 et c_3, mais elles peuvent l'être en utilisant les lignes (avec les littéraux r_1 et r_2). Les mêmes raisonnements peuvent être produits sur les lignes.

Plus généralement, étant donnée une contrainte AtMost-1 R, où l'opposé d'un littéral $r_i \in R$ implique un littéral $\neg x_i$ ($\neg r_i \rightarrow \neg x_i$), tel que ce littéral $\neg x_i$ implique un littéral b_i qui appartient à une autre contrainte AtMost-1 C ($\neg b_i \in C$), alors, en utilisant R comme contrainte de ligne et C comme contrainte de colonne, une contrainte AtMost-1 portant sur x_i peut être construite en cherchant les littéraux x_j manquants.

Pour chaque littéral a_i de la contrainte de ligne R, les littéraux x_i impliqués par $\neg r_i$ peuvent être regroupés en tant que candidats à la formation d'une ligne dans la représentation *two product*. On ne considère ici que les littéraux x_i qui impliquent un littéral différent pour la contrainte C, de manière à ce que chacun des littéraux d'une ligne correspondent à des colonnes différentes dans la matrice. Les littéraux d'une ligne forment à eux seuls une contraintes AtMost-1, qui sera étendue lors de l'étude de la ligne suivante.

Tout ceci correspond exactement à la construction de l'encodage : si un des éléments de la nouvelle contrainte est satisfait, les sélecteurs d'indices de ligne et de colonne correspondants le seront aussi. Ceci impliquera que les autres sélecteurs d'indices devront être falsifiés, tout comme les littéraux auxquels ils sont associés.

À notre connaissance, aucun algorithme capable de détecter de telles contraintes n'a encore été proposé. Nous présentons maintenant l'algorithme 1, qui est capable de détecter une approximation de l'ensemble de ces contraintes.

Algorithme 1 : ExtraisTwoProductEncoding

Entrées : L'ensemble de contraintes AtMost-1 S
Sorties : L'ensemble de contraintes S étendu

```
1 pour chaque R ∈ S faire
2     r = min(R) ;
3     l = min{l | ¬r → ¬l} ;
4     pour chaque c tel que ¬l → ¬c faire
5         pour chaque C ∈ S_c faire
6             ConstruisAtMost-1(S,R,C) ;
7 retourner S
```

Algorithme 2 : ConstruisAtMost-1

Entrées : L'ensemble de contraintes AtMost-1 S, une contrainte de ligne R, une contrainte de colonne C

```
1 A = ∅ ;                    // AtMost-1 basée sur R et C
2 pour chaque k ∈ C faire
3     hitSet = R ;           // to hit each literal once
4     pour chaque x_i tel que ¬k → ¬x_i, x_i ∉ A faire
5         pour chaque t tel que ¬x_i → ¬t faire
6             if t ∈ hitSet then
7                 hitSet ← hitSet \ {t} ;
8                 A ← A ∪ {x_i} ;
9 S ← S ∪ A ;
```

La construction d'une nouvelle contrainte AtMost-1 encodée avec le *two product encoding* passe par la recherche de deux contraintes AtMost-1 R et C qui seront les contraintes de ligne et de colonne, et qui doivent contenir respectivement les littéraux r et c,

qui doivent être utilisés en tant que sélecteurs de ligne et de colonne par des littéraux l (algorithme 1).

Nous procédons de la façon suivante : pour chaque contrainte R, on choisit un littéral r que l'on considère comme un sélecteur de ligne. Ensuite, on choisit le littéral l qui fera partie de notre nouvelle AtMost-1. Afin de réduire le temps de calcul, le littéral r choisi est le plus petit littéral de R et le littéral l est le plus petit littéral tel que $\neg r \rightarrow \neg l$ (algorithme 1, lignes 2–3). Finalement, on sélectionne la contrainte C qui contiendra les sélecteurs de colonne.

Pour chaque paire de AtMost-1 R et C, une nouvelle AtMost-1 peut être bâtie en collectant tous les littéraux x_i. Cette condition peut être vérifiée en recherchant les littéraux impliqués par le complément du littéral de sélection de colonne c : $\neg c \rightarrow \neg x_i$. De plus, un littéral x_i doit impliquer un sélecteur de ligne $r \in R$ (algorithme 2, lignes 2–4). Pour assurer cette propriété, un ensemble auxiliaire de littéraux `hitSet` est utilisé pour sauvegarder tous les littéraux sélecteurs de ligne (donc, de R) durant l'analyse de chacune des colonnes. Si, pour une colonne c, et un littéral x_i, un *nouveau* sélecteur $t \in$ `hitSet` est trouvé (algorithme 2, ligne 6), alors l'ensemble `hitSet` des littéraux est mis à jour en retirant t et la contrainte AtMost-1 en construction est mise à jour en ajoutant x_i (algorithme 2, lignes 8–9).

3.4 Détection de contraintes AtMost-2

Pour un faible nombre de littéraux et un degré faible, par exemple $k = 2$, le *naïve encoding* est compétitif. Nous présentons donc une méthode pour extraire ces contraintes.

De manière similaire à l'extraction syntaxique des contraintes AtMost-1, on analyse les contraintes ternaires à l'aide d'un algorithme glouton. Nous considérons d'abord un littéral initial n qui n'apparaît pas dans une contrainte AtMost-2 déjà extraite, on considère toutes les clauses ternaires contenant $\neg n$, et l'ensemble des littéraux candidats S est initialisé avec l'ensemble des littéraux qui apparaissent niés au moins deux fois dans ces clauses. Si, à un moment dans l'algorithme, cet ensemble contient moins de 4 littéraux, la recherche est vaine et on passe à un nouveau littéral initial. Sinon, on teste chaque triplet de littéraux de S et on regarde s'il existe dans la formule une clause ternaire les liant. Si ce test échoue, on retire de l'ensemble des candidats un des trois littéraux. Si $|S| \geq 4$ et tous les triplets de S sont soutenus par une clause, la contrainte $\sum_{l \in S} l \leq 2$ est ajoutée.

4 Détection sémantique de contraintes AtMost-k

Une autre approche qui peut être suivie pour détecter des contraintes de cardinalité est d'utiliser la propagation unitaire, dans l'esprit de [24]. L'avantage d'utiliser une approche *sémantique* plutôt qu'une approche *syntaxique* est que cela nous permet de détecter des contraintes sans nécessiter une approche spécifique pour chacun des encodages, au prix de procéder à des phases de propagation unitaires plutôt que de parcourir un NAG. Cette approche nous donne la possibilité de détecter des contraintes tant que l'encodage utilisé préserve l'arc-consistance par propagation unitaire. De plus, notre algorithme est capable de détecter des contraintes de cardinalité qui n'aurait pas été connues lors de la phase d'encodage du problème. Toutefois, les variables additionnelles ajoutées pour certains encodages peuvent altérer notre recherche et nous faire découvrir des contraintes tronquées.

Le déroulement général de notre algorithme est le suivant : en partant d'une contrainte $\sum_{i=1}^{n} l_i \leq k$, on tente de l'étendre en ajoutant de nouveaux littéraux m tels que $(\sum_{i=1}^{n} l_i) + m \leq k$.

Notre contribution est un algorithme qui détecte des contraintes de cardinalité dans les CNF en utilisant la propagation unitaire, de telle sorte que ces contraintes ne puissent plus être étendue par des littéraux.

4.1 Extension d'une contrainte de cardinalité par un littéral

L'idée de notre algorithme est la suivante : étant donnée une clause $cl = l_1 \vee l_2 \vee .. \vee l_n$, nous souhaitons vérifier que cette contrainte fasse partie d'une contrainte de cardinalité $cc = \sum_{i=1}^{n} \neg l_i + \sum_j m_j \leq n - 1$. En effet, nous savons que $cl = l_1 \vee l_2 \vee .. \vee l_n \equiv \sum_{i=1}^{n} l_i \geq 1 \equiv \sum_{i=1}^{n} \neg l_i \leq n - 1 = cc'$, et nous cherchons donc les littéraux m_j pour étendre cc'.

Retournons à notre exemple du *nested encoding* en considérant la CNF $\alpha = \neg x_1 \vee \neg x_2, \neg x_1 \vee \neg x_3, \neg x_2 \vee \neg x_3, x_3 \vee \neg x_4, x_3 \vee \neg x_5, \neg x_4 \vee \neg x_5$. $\neg x_1 \vee \neg x_2$ représente la contrainte de cardinalité $x_1 + x_2 \leq 1$. Si nous propageons un des deux littéraux x_1 ou x_2 dans α, on remarque que dans les deux cas, les littéraux $\neg x_3, \neg x_4, \neg x_5$ sont propagés. Ceci implique qu'on peut étendre $x_1 + x_2 \leq 1$ par un des littéraux x_3, x_4 ou x_5, c'est-à-dire que les contraintes $x_1 + x_2 + x_3 \leq 1$, $x_1 + x_2 + x_4 \leq 1$, $x_1 + x_2 + x_5 \leq 1$ sont des conséquences de la formule α. Plus généralement, si toutes les combinaisons valides maximales de littéraux de cc' impliquent un littéral $\neg m$, m peut être ajouté à cc'. Nous exploitons ici la propriété suivante.

Proposition 1. *Soit α une CNF. On note $\alpha(S)$ l'ensemble des littéraux propagés dans α sous les hypothèses S. Soit $cc = \sum_{i=1}^{n} l_i \leq k$ une contrainte de cardinalité, $L = \{l_i \mid 1 \leq i \leq n\}$ et $L_k = \{S | [S \subseteq L] \wedge [|S| = k]\}$. Si $\alpha \models cc$ et $\forall S \in L_k, \alpha(S) \models \neg m$, alors $\alpha \models (\sum_{i=1}^{n} l_i) + m \leq k$.*

Démonstration. Soit ω un modèle de α ,$\alpha \models \sum_{i=1}^{n} l_i \leq k$ et $\forall S \in L_k, \alpha(S) \models \neg m$. Supposons que ω ne soit pas un modèle de $\alpha \wedge (\sum_{i=1}^{n} l_i) + m \leq k$. Cela implique qu'au moins $k+1$ littéraux de $\{l_1, ..., l_n\}$ sont satisfaits. Puisque $\alpha \models \sum_{i=1}^{n} l_i \leq k$, m doit être lui aussi satisfait, ce qui est contradictoire avec l'hypothèse de départ $\forall S \in L_k, \alpha(S) \models \neg m$. □

Si plusieurs littéraux sont candidats à l'extension, il n'est pas correct de tous les ajouter d'une traite à cc'. Dans notre exemple, x_3, x_4 et x_5 sont candidats à l'extension de $x_1 + x_2 \leq 1$, mais la contrainte $x_1 + x_2 + x_3 + x_4 + x_5 \leq 1$ n'est pas conséquence de α. Nous devons aussi faire attention aux clauses unitaires de la formule originale : ces littéraux sont par définition candidats à l'extension, mais si nous en ajoutons un à notre contraintes, seuls les littéraux de ce type pourront ensuite étendre notre contrainte.

Considérons la formule $\neg x_1 \vee \neg x_2, \neg x_1 \vee \neg x_3, \neg x_3 \vee \neg x_2, \neg x_4$ et supposons que nous choisissons la clause $\neg x_1 \vee \neg x_2$ comme contrainte de cardinalité de départ ($x_1 + x_2 \leq 1$). Deux littéraux sont candidats à l'extension : x_3 et x_4. Si nous choisissons x_4, la contrainte ne sera pas plus expressive étant donné que nous savons que ce littéral doit être falsifié. Notons aussi que si la propagation unitaire amène à un conflit, il n'est pas nécessaire de filtrer les candidats étant donné que tous les littéraux sont impliqués par une formule incohérente. Dans ce cas, nous devons simplement ne pas sélectionner pour l'extension un littéral dont le complémentaire se trouverait déjà dans la contrainte de cardinalité. Cette vérification est effectuée à la ligne 1 de l'algorithme 3.

Algorithme 3 : rechercheCandidats

Entrées : une CNF α, une contrainte de cardinalité $\sum_{i=1}^{n} l_i \leq k$
Sorties : un ensemble de candidats m tels que $(\sum_{i=1}^{n} l_i) + \neg m \leq k$

```
1 candidats ← {v_i | v_i ∈ Vars(α)} ∪ {¬v_i | v_i ∈ Vars(α)} \ {¬l_i};
2 pour chaque S ⊆ {l_i} tel que |S| = k faire
3     propagés ← unitProp(α, S) ;
4     if ⊥ ∉ propagés then
5         candidats ← candidats ∩ propagés ;
6         if candidats = ∅ then
7             retourner ∅;
8 retourner candidats ;
```

L'algorithme 3 exploite la proposition 1 pour trouver le complément d'un littéral qui peut étendre une contrainte de cardinalité. L'ensemble `candidats` garde l'ensemble des candidats (les littéraux dont la négation peut étendre la contrainte). À chaque phase de propagation unitaire, seuls les littéraux propagés sont gardés. Une fois ces phases passées, seuls les littéraux propagés à chaque étape sont présents dans `candidats`, ces littéraux peuvent donc étendre la contrainte de cardinalité.

Lemme 1. *Soit α une CNF et $cc = \sum_{i=1}^{n} l_i \leq k$ une contrainte de cardinalité telle que $\alpha \models cc$. $\forall m \in$ rechercheCandidats(α,cc),$\alpha \models (\sum_{i=1}^{n} l_i) + \neg m \leq k$.*

4.2 Extension maximale d'une contrainte de cardinalité

En pratique, nous n'allons pas apprendre toutes les contraintes que nous allons découvrir, mais uniquement celles qui ne pourront plus être étendues. De plus, si une contrainte correspond en fait à une clause, nous la garderons sous forme clausale. L'algorithme 3 calcule l'ensemble des littéraux propagés par tous les ensembles L_k. Si l'ensemble retourné par cet algorithme est vide, on ne peut plus étendre notre contrainte ; s'il est en revanche non-vide, il est possible d'étendre notre contrainte avec un des littéraux de l'ensemble.

Lemme 2. *Soit α une CNF. $\forall c \in \alpha$, $\alpha \models$ étendsCardDepuisClause(α, c).*

De manière itérative, nous recherchons un littéral candidat à l'extension, nous l'ajoutons à la contrainte, et répétons ces étapes tant qu'un candidat est trouvé. À partir de la seconde itération, il n'est plus nécessaire de calculer tous les ensembles L_k, puisque certain auront été calculés lors d'itérations antérieures. Plus précisément, pour trouver le n^{me} d'une contrainte, nous avons déjà calculé $\binom{n-1}{k}$ des $\binom{n}{k}$ des phases de propagations nécessaires pour l'appel courant de l'algorithme 3. Les seules phases manquantes sont celles où figure parmi les hypothèses le dernier candidat ajouté. Ce processus est décrit dans l'algorithme 4.

Grâce à cette remarque, nous proposons l'algorithme 5 qui calcule efficacement des contraintes de cardinalité maximales.

Le calcul des $\binom{n}{k}$ phases de propagation unitaire est la partie difficile de l'algorithme. En considérant que le coût d'une de ces phases est borné par le nombre de variables, le lemme suivant est valide.

Lemme 3. *Soit α une CNF avec n variables et l littéraux. Soit c une clause de α de taille $|c| = k+1$. étendsCardDepuisClause(α, c) a une complexité en $O(\binom{n}{k} \times l)$.*

Algorithme 4 : raffineCandidats

Entrées : une CNF α, une contrainte $\sum_{i=1}^{n} l_i + l_{new} \leq k$, un ensemble de littéraux L

Sorties : un ensemble de candidats m tels que $(\sum_{i=1}^{n} l_i) + l_{new} + \neg m \leq k$

```
candidats ← L;
pour chaque S' = S ∪ {l_new} tel que S ⊆ {l_i} et |S| = k − 1 faire
    propagés ← unitProp(α, S') ;
    si ⊥ ∉ propagés alors
        candidats ← candidats ∩ propagés ;
        si candidats = ∅ alors
            retourner ∅ ;
retourner candidats ;
```

Algorithme 5 : étendsCardDepuisClause

Entrées : une CNF α, une clause c

Sorties : une contrainte de cardinalité cc ou c

```
cc ← ∑_{l∈c} ¬l ≤ |c| − 1 ;
candidats ← rechercheCandidats(α, cc) ;
tant que candidats ≠ ∅ faire
    sélectionner m dans candidats ;
    cc ← ∑_{l_i∈cc} l_i + ¬m ≤ |c| − 1 ;
    candidats ← raffineCandidats(α, cc, candidats \ {m}) ;
si |cc| > |c| alors retourner cc ;
sinon retourner c ;
```

4.3 Retrait des clauses dominées

La dernière étape dans notre approche consiste à détecter les clauses qui sont conséquences de contraintes découvertes, et à les retirer dans une optique d'efficacité du solveur. Le but est d'obtenir une formule équivalente à l'originale en retirant un maximum de clauses qui seraient devenues inutiles suite à la découverte de contraintes de cardinalité.

Nous utilisons pour cela la règle décrite dans [8], utilisée dans 3MCard [32], pour déterminer si une clause (exprimée sous la forme d'une contrainte AtMost-1) est dominée par une contrainte découverte. Cette règle indique que, étant données deux contraintes de cardinalité, $L \geq d$ domine $L' \geq d'$ si et seulement si $|L \setminus L'| \leq d - d'$. Ainsi, afin de tenter d'étendre une contrainte sous forme clausale, on utilise cette règle pour déterminer si elle n'est pas dominée par une contrainte révélée. Dans ce cas, nous ne traitons pas cette contrainte et nous la retirons du problème. Nous retirons aussi du problème les clause qui ont été étendues.

Un autre point important pour éviter la redondances de contraintes est de considérer en premier les clauses les plus petites comme candidates à l'extension. De ce fait, nous découvrons des contraintes dont les degrés sont croissants, et une contrainte découverte ne peut donc pas être conséquence d'une contrainte découverte antérieurement si les littéraux sont communs.

Algorithme 6 : détecterContraintesDansCNF

Entrées : une CNF α et une borne k

Sorties : une formule $\phi \equiv \alpha$ contenant des contraintes de cardinalité de degré $\leq k$ et des clauses

```
φ ← ∅ ;
pour chaque clause c ∈ α de taille |c| croissante telle que |c| ≤ k + 1 faire
    si aucune contrainte révélée cc ∈ φ ne domine c alors
        φ ← φ ∪ étendCardDepuisClause(α, c) ;
retourner φ ;
```

Puisque les nouvelles contraintes sont conséquences de la formule, et que les contraintes enlevées sont conséquences de contraintes ajoutées, l'algorithme assure que la nouvelle formule est logiquement équivalente à l'originale. D'où le théorème.

Theorem 1. *Soit α une CNF et k un entier donné.* $\alpha \equiv$ *détecterContraintesDansCNF*(α, k)

Dans notre exemple sur le *nested encoding*, notre approche fonctionne comme suit. Tout d'abord, nous tentons d'étendre $x_1 + x_2 \leq 1$. Nous déterminons qu'elle peut être étendue en $x_1 + x_2 + x_3 \leq 1$ ou $x_1 + x_2 + x_4 + x_5 \leq 1$. Supposons que la deuxième est trouvée. La CNF est réduite en enlevant les clauses dominées par cette contrainte, à savoir $\neg x_1 \vee \neg x_2$ et $\neg x_4 \vee \neg x_5$. On cherche ensuite à étendre $x_1 + \neg x_3 \leq 1$, on obtient la contrainte $x_1 + x_2 + x_3 \leq 1$. Ceci nous permet de retirer la clause dominée $\neg x_2 \vee \neg x_3$. Ensuite, $\neg x_3 + x_4 \leq 1$ est étendue en $\neg x_3 + x_4 + x_5 \leq 1$. Les clauses restantes, toutes dominée, sont retirées de la formule, et l'algorithme s'arrête puisqu'il n'y a plus de clauses à étudier. Notons que si la première contrainte découverte avait été $x_1 + x_2 + x_3 \leq 1$, l'algorithme n'aurait pas été capable de révéler $x_1 + x_2 + x_4 + x_5 \leq 1$ car toutes les clauses contenant $\neg x_1$ auraient été retirées.

En ce qui concerne la complexité de l'algorithme, le pire cas est atteint lorsqu'on cherche à étendre toutes les clauses, ce qui implique le lemme suivant.

Lemme 4. *Soit α une CNF avec n variables, m clauses and l littéraux. Soit k un entier telle que $0 < k \leq n$ et $m_k \leq m$ le nombre de clauses de taille $\leq k + 1$ dans α. détecterContraintesDansCNF(α, k) a une complexité en $O(m_k \times \binom{n}{k} \times l)$.*

5 Résultats expérimentaux

Les résultats expérimentaux montrent que les méthodes que nous avons proposées détectent un nombre

important de contraintes de cardinalité. Cette analyse a été faite sur des benchmarks académiques, comme des grilles de Sudoku ou des instances du problème des pigeons, pour lesquels nous connaissons le nombre de contraintes à découvrir dans la CNF, et qui sont simples à résoudre en utilisant la résolution généralisée quand les contraintes sont des contraintes de cardinalité. Nos expérimentations ont été réalisées sur des machines dont les processeurs sont des Intel Xeon (@2.66GHz) avec 32Go de RAM ; les temps limites d'exécution ont été fixés à 900s.

L'approche statique est implémentée dans la dernière version de Lingeling. L'approche syntaxique associée à la reconnaissance du two product encoding, ainsi que l'approche sémantique, ont été implémentées dans Riss. Puisque Riss ne sait pas tirer avantage des contraintes de cardinalité, nous l'avons utilisé comme un préprocesseur rapide pour fournir le nouveau problème à Sat4j, qui utilise la résolution généralisée. Cela nous permet de vérifier si les contraintes détectées sont suffisantes pour résoudre ces benchmarks. Nous avons aussi comparé nos approches avec SBSAT [1].

5.1 Problèmes des pigeons

Ces benchmarks sont connus pour être extrêmement difficiles pour les solveurs basés sur la résolution [19]. Pour $n+1$ pigeons et n pigeonniers, le problème est d'affecter à chaque pigeon un pigeonnier en ayant un pigeon par case au maximum. Chaque variable booléenne $x_{i,j}$ représente l'information "le pigeon i est dans le pigeonnier j". Le problème est représenté à l'aide de $n+1$ clauses $\bigvee_{j=1}^{n} x_{i,j}$ et n contraintes de cardinalité $\sum_{j=1}^{n+1} x_{i,j} \leq 1$. Nous avons généré ces benchmarks pour n allant de 10 à 15, et n allant de 25 à 200 par pas de 25 en utilisant six encodages différents : binomial, product, binary, ladder, commander et sequential. Les résultats sont donnés dans la figure 2.

Comme nous pouvions nous y attendre, Sat4j est incapable de résoudre la plupart des benchmarks quand les problèmes sont sous forme clausale. La détection sémantique découvre beaucoup de contraintes et permet au solveur de résoudre plus d'instances que les autres solveurs (ceci est particulièrement vrai pour les encodages*pairwise*, sequential et commander). Notons que l'instance *pairwise* pour $n = 200$ contient 402000 variables et 4020201 clauses, ce qui semble démontrer que notre approche résiste au passage à l'échelle. L'analyse statique associée à la reconnaissance du *two product encoding* se débrouille elle-aussi très bien sur la plupart des encodages, et est la meilleure pour le *two product encoding*, comme attendu. Les encodages *binary* et *ladder* sont plus difficiles à détecter avec nos approches. SBSAT est bien moins efficace (excepté pour le *ladder*). Lingeling n'est efficace que sur le *pairwise encoding*, comme nous l'avions prévu.

5.2 Benchmarks hautement combinatoires

Ces benchmarks incohérents de *balanced block design* sont décrits dans [31, 16] ; ils contiennent des contraintes de cardinalité AtMost-2. Nous utilisons les benchmarks soumis à la compétition SAT09 (alors appelés *sgen*) qui étaient les plus petites instances insatisfiables de benchmarks hautement combinatoires, ainsi que des benchmarks fournis par Jakob Nordström et Mladen Miksa de KTH [28]. Nos deux approches permettent de récupérer toutes les contraintes et de résoudre de manière quasiment instantanée tous ces benchmarks, comme montré à la figure 3.

Les solveurs Lingeling et Sat4j sans prétraitement, qui n'utilisent pas la détection de contraintes de cardinalité, montrent que ces fichiers de test sont trop difficiles pour les solveurs actuels.

5.3 Benchmarks de Sudoku

Les benchmarks de Sudoku contiennent uniquement des contraintes de cardinalité $= 1$ représentées par une clause et une contrainte AtMost-1. Ces instances sont triviales à résoudre ; leur intérêt vient du fait que les différentes contraintes de cardinalité partagent un grand nombre de littéraux, ce qui pourrait gêner nos opérations de prétraitement. Nous utilisons deux instances de grilles vides $n \times n$, pour $n = 9$ et $n = 16$. La grille contient $n^2 \times 4$ contraintes AtMost-1 : pour $n = 9(16)$, il y a 324(1024) contraintes (toutes encodées *via* le pairwise encoding.

Les contraintes AtMost-1 détectées par l'approche syntaxique contiennent bien toutes 9 (resp. 16) littéraux, cependant certaines sont manquantes : 300/324 (resp. 980/1024) contraintes ont été découvertes par cette approche. L'approche sémantique, quant à elle, découvre toutes les contraintes. Cela vient du fait que toutes les contraintes possèdent au moins une clause binaire qui leur est propre, et qui n'est donc pas retirée quand une autre contrainte a été trouvée. Cette clause est ensuite étendue pour donner la contrainte à laquelle elle appartient.

6 Conclusion

Nous avons présenté deux approches capables de retrouver des contraintes de cardinalité dans des CNF. La première est basée sur une analyse syntaxique du NAND graph, une structure de données utilisée

1. Nous avons tenté d'inclure 3MCard dans nos résultats, mais il n'a malheureusement pas été capable de lire ou de résoudre la plupart de nos fichiers de test

Prétraitement Solver	#inst.	Lingeling Lingeling	Synt.(Riss) Sat4jCP	Sem.(Riss) Sat4jCP	∅ SBSAT	∅ Sat4jCP
Pairwise	14	**14 (3s)**	13 (244s)	14 (583s)	6 (0s)	1 (196s)
Binary	14	3 (398s)	2 (554s)	**7 (6s)**	6 (7s)	2 (645s)
Sequential	14	0 (0s)	**14 (50s)**	**14 (40s)**	10 (6s)	1 (37s)
Product	14	0 (0s)	**14 (544s)**	11 (69s)	6 (25s)	2 (346s)
Commander	14	1 (3s)	7 (0s)	**14 (40s)**	9 (187s)	1 (684s)
Ladder	14	0 (0s)	11 (505s)	11 (1229s)	**12 (26s)**	1 (36s)

FIGURE 2 – Six encodage du problème des pigeonniers : nombre d'instances résolues et somme des temps de calcul (prétraitement et solver) pour les instances résolues entre parenthèses.

Prétraitement Solver	#inst.	Lingeling Lingeling	Synt.(Riss) Sat4jCP	Sem.(Riss) Sat4jCP	∅ SBSAT	∅ Sat4jCP
Sgen unsat	13	0 (0s)	**13 (0s)**	**13 (0s)**	9 (614s)	4 (126s)
Fixed bandwidth	23	2 (341s)	**23 (0s)**	**23 (0s)**	**23 (1s)**	13 (1800s)
Rand. orderings	168	16 (897s)	**168 (7s)**	**168 (8s)**	99 (2798s)	69 (3541s)
Rand. 4-reg.	126	6 (1626s)	**126 (4s)**	**126 (5s)**	84 (2172s)	49 (3754s)

FIGURE 3 – Plusieurs familles de problèmes hautement combinatoires : nombre d'instances résolues et somme des temps de calcul (prétraitement et solver) pour les instances résolues entre paranthèses.

dans certains solveurs pour gérer les contraintes binaires de manière efficace, et permet de récupérer des contraintes AtMost-1 et AtMost-2. La seconde, basée sur la propagation unitaire dans la formule, et un algorithme générique capable de retrouver des contraintes AtMost-k de tailles quelconques. Nous montrons que ces deux approches sont capables de retrouver des contraintes dans des problèmes en CNF connus pour être hautement combinatoires. Nos expérimentations suggèrent que l'approche syntaxique est particulièrement utile en ce qui concerne les contraintes AtMost-1 et AtMost-2, quand l'approche sémantique semble plus robuste au sens où elle permet de récupérer des contraintes de degré arbitraire.

Nos approches sont utiles pour traiter les plus petits benchmarks UNSAT non-résolus de la compétition SAT2009 (sgen), qui sont toutes résolues par la version de Sat4j basée sur la résolution généralisée dans la seconde une fois que les contraintes AtMost-2 ont été mises à jour (ceci avait déjà été remarqué dans[34]). La différence entre nos deux approches est néanmoins visible avec le benchmark-challenge de [16], qu'aucun solveur n'avait pu résoudre en l'espace d'une journée. Cette instance est résolue en une seconde, après avoir révélé les 22 contraintes AtMost-2 et les 20 contraintes AtMost-3 grâce à l'approche sémantique (l'approche syntaxique n'est pas capable de retrouver les contraintes AtMost-3).

Nous avons aussi été capables de révéler des contraintes de cardinalité sur de grands benchmarks applicatifs. Cependant, l'utilisation de ces informations pour améliorer le temps de calcul des solveurs est laissé à un travail futur. Vérifier que ces contraintes étaient connus au moment de l'encodage, ou déterminer que ces contraintes étaient "cachées" est un problème ouvert. Une question intéressante, bien qu'en dehors du domaine de ce papier, serait d'étudier le système de preuve résultant de la combinaison de notre approche sémantique (règles d'extension et de domination) et de la résolution généralisée.

Références

[1] Fadi A. Aloul, Arathi Ramani, Igor L. Markov, and Karem A. Sakallah. Generic ilp versus specialized 0-1 ilp : an update. In *ICCAD*, pages 450–457, 2002.

[2] Carlos Ansótegui, Jose Larrubia, Chu Min Li, and Felip Manyà. Exploiting multivalued knowledge in variable selection heuristics for sat solvers. *Ann. Math. Artif. Intell.*, 49(1-4) :191–205, 2007.

[3] Carlos Ansótegui and Felip Manyà. Mapping problems with finite-domain variables to problems with boolean variables. In *SAT (Selected Papers*, pages 1–15, 2004.

[4] Carlos José Ansótegui Gil. *Complete SAT solvers for Many-Valued CNF Formulas.* PhD thesis, University of Lleida, 2004.

[5] Roberto Asín, Robert Nieuwenhuis, Albert Oliveras, and Enric Rodríguez-Carbonell. Cardinality networks and their applications. In *SAT*, pages 167–180, 2009.

[6] Olivier Bailleux and Yacine Boufkhad. Efficient cnf encoding of boolean cardinality constraints. In *CP*, pages 108–122, 2003.

[7] Pedro Barahona, Steffen Hölldobler, and Van-Hau Nguyen. Representative Encodings to Translate Finite CSPs into SAT. In *CPAIOR 2014, à paraître*, 2014.

[8] Peter Barth. Linear 0-1 inequalities and extended clauses. In *LPAR*, pages 40–51, 1993.

[9] Yael Ben-Haim, Alexander Ivrii, Oded Margalit, and Arie Matsliah. Perfect hashing and cnf encodings of cardinality constraints. In *SAT*, pages 397–409, 2012.

[10] Armin Biere. Lingeling, plingeling and treengeling entering the sat competition 2013. In *SAT Competition 2013; Solver and Benchmark Descriptions*, pages 51–52, 2013.

[11] Jing-Chao Chen. A new sat encoding of the at-most-one constraint. In *10th Int. Workshop of Constraint Modelling and Reformulation*, 2010.

[12] W. Cook, C.R. Coullard, and Gy. Turán. On the complexity of cutting-plane proofs. *Discrete Applied Mathematics*, 18(1) :25 – 38, 1987.

[13] Niklas Eén and Niklas Sörensson. Translating pseudo-boolean constraints into sat. *JSAT*, 2(1-4) :1–26, 2006.

[14] Alan M. Frisch and Paul A. Giannaros. Sat encodings of the at-most-k constraint : Some old, some new, some fast, some slow. In *ModRef 2010*, 2010.

[15] Zhaohui Fu and Sharad Malik. Extracting logic circuit structure from conjunctive normal form descriptions. In *VLSI Design*, pages 37–42, 2007.

[16] Allen Van Gelder and Ivor Spence. Zero-one designs produce small hard sat instances. In *SAT*, pages 388–397, 2010.

[17] Ian P Gent and Peter Nightingale. A new encoding of alldifferent into sat. *Proc. 3rd International Workshop on Modelling and Reformulating Constraint Satisfaction Problems*, pages 95–110, 2004.

[18] I.P. Gent and B.M. Prosser, P. andSmith. A 0/1 encoding of the gaclex constraint for pairs of vectors. In *ECAI 2002 workshop W9 : Modelling and Solving Problems with Constraints*. University of Glasgow, 2002.

[19] Armin Haken. The intractability of resolution. *Theoretical Computer Science*, 39(0) :297 – 308, 1985.

[20] Steffen Hölldobler and Van Hau Nguyen. On SAT-Encodings of the At-Most-One Constraint. In George Katsirelos and Claude-Guy Quimper, editors, *Modref 2013*, pages 1–17, 2013.

[21] J. N. Hooker. Generalized resolution and cutting planes. *Ann. Oper. Res.*, 12(1-4) :217–239, 1988.

[22] RichardM. Karp. Reducibility among combinatorial problems. In RaymondE. Miller, JamesW. Thatcher, and JeanD. Bohlinger, editors, *Complexity of Computer Computations*, The IBM Research Symposia Series, pages 85–103. Springer US, 1972.

[23] W. Klieber and G. Kwon. Efficient cnf encoding for selecting 1 from n objects. In *International Workshop on Constraints in Formal Verification*, 2007.

[24] Daniel Le Berre. Exploiting the real power of unit propagation lookahead. *Electronic Notes in Discrete Mathematics*, 9 :59–80, 2001.

[25] Norbert Manthey, Marijn Heule, and Armin Biere. Automated reencoding of boolean formulas. In *Haifa Verification Conference*, pages 102–117, 2012.

[26] Norbert Manthey and Peter Steinke. Quadratic direct encoding vs. linear order encoding, a one-out-of-n transformation on cnf. In *First International Workshop on the Cross-Fertilization Between CSP and SAT (CSPSAT11)*, 2011.

[27] Ruben Martins, Vasco M. Manquinho, and Inês Lynce. Parallel search for maximum satisfiability. *AI Commun.*, 25(2) :75–95, 2012.

[28] Mladen Miksa and Jakob Nordstrom. Long proofs of (seemingly) simple formulas. In *Proc. of SAT14*, 2014. à paraître.

[29] Richard Ostrowski, Éric Grégoire, Bertrand Mazure, and Lakhdar Sais. Recovering and exploiting structural knowledge from cnf formulas. In *CP*, pages 185–199, 2002.

[30] Carsten Sinz. Towards an optimal cnf encoding of boolean cardinality constraints. In *CP*, pages 827–831, 2005.

[31] Ivor Spence. sgen1 : A generator of small but difficult satisfiability benchmarks. *ACM Journal of Experimental Algorithmics*, 15, 2010.

[32] Martijn van Lambalgen. 3mcard 3mcard a lookahead cardinality solver. Master's thesis, Delft University of Technology, 2006.

[33] Toby Walsh. Sat v csp. In *CP*, pages 441–456, 2000.

[34] Sean Weaver. *Satisfiability Advancements Enabled by State Machines*. PhD thesis, University of Cincinnati, 2012.

Classification non supervisée mono et bi-objectif par la programmation par contraintes

Thi-Bich-Hanh Dao, Khanh-Chuong Duong, Christel Vrain

Univ. Orléans, INSA Centre Val de Loire, LIFO EA 4022, F-45067, Orléans, France
{thi-bich-hanh.dao, khanh-chuong.duong, christel.vrain}@univ-orleans.fr

Résumé

La classification non supervisée est une tâche importante dans le domaine de la Fouille de Données. Elle est souvent représentée par un problème d'optimisation. L'optimisation peut être mono-objectif, e.g. minimisation du diamètre maximal des clusters D, maximisation de la marge entre clusters S ; ou bi-objectif $(\max S, \min D)$. Nous proposons un nouveau modèle pour la classification non supervisée avec ces différents critères en Programmation par Contraintes. Notre modèle permet une extension directe vers la classification non supervisée sous contraintes utilisateur, car différents types de contraintes utilisateur connus sont intégrés directement. Nous illustrons la déclarativité de notre modèle en l'appliquant au clustering bi-objectif. Des expériences sur des ensembles de données classiques montrent l'intérêt de notre modèle.

Abstract

Cluster analysis is an important task in Data Mining. It is usually formalized by an optimization problem with a single criterion, such as minimizing the maximal cluster diameter D or maximizing the split between clusters S, or with two criteria $(\max S, \min D)$. We propose a Constraint Programming framework for cluster analysis with these different criteria. Our framework extends directly to constrained clustering since user-defined constraints are easily integrated. Thanks to the declarativity of our approach, bi-objective clustering can be easily implemented in our framework. Experiments on classical datasets show the interest of our framework.

1 Introduction

La classification non supervisée (ou clustering) est une tâche importante en Fouille de Données. Elle est souvent représentée par un problème d'optimisation. De nombreux travaux ont été réalisés pour cette tâche pour différents critères d'optimisation. Depuis les années 2000, des contraintes utilisateur sont introduites pour guider la recherche et pour atteindre les solutions souhaitées s'il en existe. L'extension avec des contraintes utilisateur se fait soit en adaptant les algorithmes classiques pour gérer les contraintes, soit en modifiant la distance entre données pour traduire les contraintes. Cependant, la plupart des approches ne garantissent pas la satisfaction de toutes les contraintes ou la qualité de la solution vis à vis d'un critère. De plus, chaque algorithme est construit spécifiquement pour un critère ou pour un type de contraintes utilisateur. Dans un travail récent [5], nous proposons un cadre général basé sur la Programmation par Contraintes pour modéliser le clustering sous contraintes utilisateur. Ce cadre intègre différents critères : minimiser le diamètre maximal des clusters, maximiser la marge minimale entre clusters ou minimiser la somme des dissimilarités intra-cluster. Différents types de contraintes utilisateur sont également intégrés. Le critère d'optimisation considéré est mono-critère. Cependant le clustering avec le critère du diamètre souffre souvent d'un effet de découpage [4], *i.e.* des objets assez similaires peuvent être classés dans des classes différentes pour réduire le diamètre, alors que le clustering avec le critère de la marge souffre d'un effet de chaîne [14], *i.e.* des objets très différents peuvent se trouver dans une même classe. Considérer ensemble ces deux critères permet de remédier à ces problèmes. Il existe des algorithmes exacts [10] ou approchés [23, 21] traitant cette optimisation bi-critère, mais aucun ne considère des contraintes utilisateur.

Dans ce papier, nous présentons un nouveau modèle en Programmation par Contraintes qui traite le clustering sous contraintes utilisateur avec l'optimisation mono-critère du diamètre et de la marge. Ce nouveau modèle est sensiblement différent du modèle présenté

dans [5] dans le choix de variables et de contraintes. Il est plus efficace tout en étant plus flexible, car alors que le modèle précédent exigeait que le nombre de clusters k soit fixé à l'avance, dans ce nouveau modèle, le nombre k est seulement borné par k_{min} et k_{max}. Etant déclaratif et général, notre modèle peut être utilisé directement pour traiter le clustering bi-critère marge-diamètre. Nous présentons cette application en utilisant deux méthodes, dont l'une par des optimisations mono-critère successives et l'autre par l'utilisation de la contrainte globale *Pareto*. Des expérimentations sur des bases de données classiques montrent qu'avec notre modèle nous pouvons obtenir des algorithmes plus efficaces que les algorithmes exacts existants.

Nous présentons dans la section 2 les notions préliminaires sur le clustering sous contraintes et sur l'optimisation bi-critère, ainsi que les travaux connexes. Nous présentons le nouveau modèle dans la section 3 et l'application pour le clustering bi-critère dans la section 4. Les expérimentations sont présentées dans la section 5 et une discussion sur les travaux futurs et la conclusion sont dans la section 6.

2 Préliminaires

2.1 Classification non supervisée

La classification non supervisée, souvent appelée par son équivalent anglais clustering, est une tâche de Fouille de Données qui a pour objectif de regrouper des données en un ensemble de classes ou clusters, de façon à ce que les objets d'une même classe ont une forte similarité entre eux et diffèrent fortement des objets des autres classes. La classification non supervisée est souvent vue comme un problème d'optimisation, i.e., trouver une partition des objets qui optimise un critère donné. Nous considérons une base composée de n objets $\mathcal{O} = \{o_1, \ldots, o_n\}$ et une mesure de dissimilarité $d(o_i, o_j)$ entre deux objets o_i et o_j de $\mathcal{O}$. Une partition Δ des objets en k classes $C_1, \ldots, C_k$ est telle que : (1) pour tout $c \in [1, k]$, $C_c \neq \emptyset$, (2) $\cup_c C_c = \mathcal{O}$ et (3) pour tout $c \neq c'$, $C_c \cap C_{c'} = \emptyset$. Le critère à optimiser peut être entre autres la minimisation du diamètre maximal des clusters ou la maximisation de la marge minimale entre clusters.

Définition 2.1. Le diamètre maximal des clusters d'une partition Δ est la plus grande distance entre deux objets d'une même classe :

$$D(\Delta) = \max_{c \in [1,k], o_i, o_j \in C_c} (d(o_i, o_j)).$$

Définition 2.2. La marge minimale entre clusters d'une partition Δ est la plus petite des distances entre deux objets de différentes classes :

$$S(\Delta) = \min_{1 \leq c < c' \leq k, o_i \in C_c, o_j \in C_{c'}} (d(o_i, o_j)).$$

La plupart des algorithmes de classification non supervisée reposent sur un critère d'optimisation, et pour des raisons de complexité ne cherchent qu'un optimum local. Plusieurs optima peuvent exister, certains pouvant être plus proches de celui recherché par l'utilisateur. Afin de mieux modéliser la tâche d'apprentissage, mais aussi dans l'espoir de réduire la complexité, des contraintes définies par l'utilisateur peuvent être ajoutées. On parle alors de classification non supervisée sous contraintes dont le but est de trouver des clusters satisfaisant les contraintes utilisateur. Ces contraintes peuvent porter sur les classes (contraintes de classe) ou sur les objets (contraintes objet). La plupart des travaux ont porté sur les contraintes objets, introduites dans [19]. Deux types de contraintes objets sont couramment utilisés : must-link ou cannot-link. Une contrainte must-link spécifie que deux objets o_i et o_j doivent apparaître dans la même classe :

$$\forall c \in [1, k],\ o_i \in C_c \Leftrightarrow o_j \in C_c.$$

Une contrainte cannot-link spécifie que deux objets ne doivent pas être dans la même classe :

$$\forall c \in [1, k],\ \neg(o_i \in C_c \wedge o_j \in C_c).$$

Les contraintes de classe imposent des restrictions sur les classes. La contrainte de capacité minimale (resp. maximale) exige que chaque classe ait au moins (resp. au plus) un nombre donné α (resp. β) d'objets : $\forall c \in [1, k],\ |C_c| \geq \alpha$ (resp. $\forall c \in [1, k],\ |C_c| \leq \beta$). La contrainte de diamètre maximal spécifie une borne supérieure γ sur le diamètre de chaque cluster :

$$\forall c \in [1, k], \forall o_i, o_j \in C_c,\ d(o_i, o_j) \leq \gamma.$$

La contrainte de marge minimale, aussi appelée δ-contrainte dans [8], spécifie une borne inférieure δ sur la marge entre deux classes :

$$\forall c \in [1, k], \forall c' \neq c, \forall o_i \in C_c, o_j \in C_{c'}, d(o_i, o_j) \geq \delta.$$

Dans les dix dernières années, beaucoup de travaux ont étendu les algorithmes classiques pour traiter des contraintes must-link et cannot-link, comme par exemple une extension de COBWEB [19], des k-moyennes [20, 2], de la classification hiérarchique [7] ou de la classification spectrale [16, 22]. Ceci est effectué en modifiant soit la mesure de dissimilarité, soit la fonction objectif à optimiser ou encore la stratégie de recherche. A notre connaissance, il n'y a pas de solution générale pour étendre les algorithmes classiques à différents types de contraintes. Notre modèle fondé sur la Programmation par Contraintes permet d'ajouter directement des contraintes utilisateur.

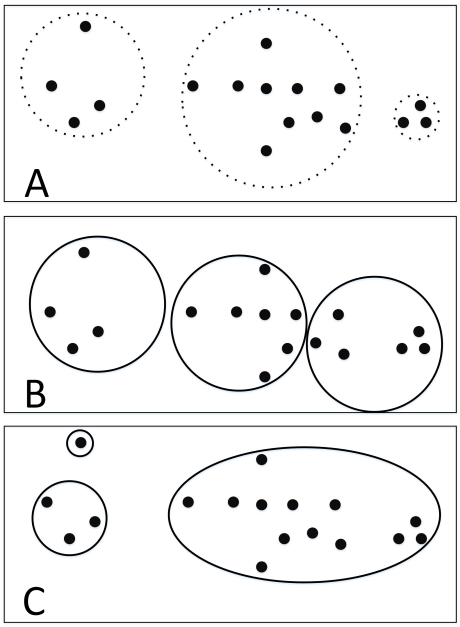


FIGURE 1 – Effets de critère : (A) jeu de données, (B) critère du diamètre, (C) critère de la marge.

2.2 Classification non supervisée bi-critère

Le critère de diamètre vise à trouver des clusters compacts mais souffre souvent de l'effet de découpage [4], *i.e.* des objets assez similaires peuvent être classés dans des classes différentes pour réduire le diamètre. Par contre, le critère de la marge peut souffrir de l'effet de chaîne [14], *i.e.* des objets très différents peuvent se trouver dans une même classe. La figure 1 montre un exemple de ces effets. L'image A montre trois groupes clairement reconnaissables. L'image B montre la solution obtenue avec le critère de diamètre si le nombre de classes est égal à trois. Dans cette partition, il y a des points qui sont très proches mais qui sont classés dans deux groupes différents. Si le critère est de maximiser la marge minimale, on obtient la partition présentée dans l'image C. À cause de l'effet de chaîne, le plus grand groupe contient des points très éloignés.

Pour éviter ces problèmes, on peut soit utiliser les contraintes utilisateur, soit considérer ces critères en même temps. Considérant les deux critères ensemble, une partition idéale aurait un diamètre minimum et une marge maximum. Malheureusement, une telle solution en général n'existe pas car ces deux critères sont souvent conflictuels. A la place, on devra trouver les solutions de Pareto. Une solution de Pareto est une solution pour laquelle il est impossible d'améliorer un objectif sans détériorer l'autre objectif.

Définition 2.3. [10] Considérons le bi-critère de maximiser la marge entre clusters et de minimiser le diamètre maximal des clusters. Une partition Δ' domine une partition Δ si et seulement si : $D(\Delta') \leq D(\Delta)$ et $S(\Delta') > S(\Delta)$ ou $D(\Delta') < D(\Delta)$ et $S(\Delta') \geq S(\Delta)$. Une partition Δ est un optimal de Pareto si et seulement s'il n'existe pas d'autre partition Δ' qui domine Δ. Deux optima de Pareto Δ et Δ' sont équivalents si et seulement si $D(\Delta) = D(\Delta')$ et $S(\Delta) = S(\Delta')$. Un ensemble $\mathcal{A}$ d'optima de Pareto est complet si et seulement si chaque optimal de Pareto Δ tel que $\Delta \notin \mathcal{A}$ est équivalent à une partition $\Delta' \in \mathcal{A}$, et $\mathcal{A}$ est minimal s'il n'existe pas dans $\mathcal{A}$ deux partitions qui sont équivalentes.

Les partitions montrées dans la figure 1 font partie des solutions de Pareto (données dans la section 5) pour le bi-critère $(\max S, \min D)$. Parmi les solutions de Pareto, une sera sélectionnée si l'utilisateur fixe un critère sur les objectifs, par exemple $\max(S/D)$ ou $\min[\alpha D - (1-\alpha)S]$ avec $0 \leq \alpha \leq 1$. La partition de l'image A est celle qui maximise le ratio S/D.

2.3 Travaux connexes

Chercher une partition maximisant la marge entre clusters est un problème polynomial [10], mais qui devient NP-difficile avec des contraintes utilisateur comme par exemple des contraintes cannot-link [9]. Concernant la minimisation du diamètre maximal, le problème est polynomial pour $k = 2$, mais devient NP-difficile dès que $k \geq 3$ [13]. Les algorithmes utilisés sont souvent heuristiques, méta-heuristiques ou des algorithmes d'approximation. Un algorithme polynomial 2-approximation FPF (Furthest Point First) est proposé dans [12]. Si D^* est le diamètre de la partition optimale, l'algorithme garantit de trouver une partition Δ tel que $D^* \leq D(\Delta) \leq 2D^*$, si la mesure utilisée satisfait l'inégalité triangulaire. De plus, il a été prouvé que trouver Δ tel que $D(\Delta) \leq (2-\epsilon)D^*$ est NP-difficile pour tout $\epsilon > 0$, lorsque les points sont dans un espace à 3 dimensions. A notre connaissance, il existe peu d'algorithmes exacts. Un algorithme exact basé sur la coloration de graphes est proposé dans [13]. Avec une base de données de n points, la valeur du diamètre optimal D^* se trouve dans les $n(n-1)/2$ distances disponibles. L'algorithme trie ces distances dans l'ordre décroissant et vérifie pour chaque distance si elle peut être le diamètre maximal. Le test de satisfaction est basé sur la coloration de graphes. La première distance qui satisfait le test est le diamètre optimal. Une autre approche exacte utilise la recherche *branch-and-bound* [3]. Cette approche utilise un algorithme hiérarchique pour trouver une borne supérieure et une stratégie de réordonnancement des points pour

réduire l'espace de recherche. Cependant, il faut noter qu'il n'existe pas d'algorithmes exacts pour ces critères simples en présence de contraintes utilisateurs.

Concernant l'optimisation bi-critère diamètre-marge, un algorithme pour trouver un ensemble complet et minimal de solutions de Pareto est proposé dans [10]. Il est prouvé que pour n points, quel que soit k, quelle que soit la partition, le nombre des valeurs possibles pour la marge minimale est au plus $n-1$. Pour trouver ces valeurs, on considère le graphe complet où les points sont des sommets et le poids de chaque arête $\{i,j\}$ est $d(i,j)$. Un arbre couvrant de poids minimal du graphe est calculé et les poids des arêtes de l'arbre forme l'ensemble des valeurs possibles pour la marge. Pour n points il existe toujours $n(n-1)/2$ valeurs possibles pour le diamètre. Ces valeurs pour la marge et pour le diamètre sont considérées dans l'ordre décroissant. Pour chaque valeur pour la marge m, les points i,j ayant $d(i,j) < m$ sont fusionnés et la coloration de graphe est ensuite utilisée pour tester s'il existe une partition en k groupes avec une valeur de diamètre d. L'algorithme calcule l'ensemble des solutions de Pareto pour différentes valeurs de k.

Dans le cas de bi-partition ($k=2$), un algorithme polynomial exact est proposé dans [23, 21]. Pour $k>2$, en utilisant l'algorithme FPF, [21] propose un algorithme 2-approximation. Ces deux algorithmes sont basés sur l'approche de [10] : la construction d'un arbre couvrant est utilisée pour trouver les valeurs pour la marge et la coloration de graphe est utilisée pour tester les valeurs pour le diamètre. Il est toujours à noter que ces approches considérant le bi-critère ne considèrent pas des contraintes utilisateur.

Pour l'optimisation bi-critère (obj_1, obj_2) en général, un algorithme est proposé dans [24]. Cet algorithme choisit un objectif, par exemple obj_2, et réalise successivement l'optimisation avec cet objectif, à chaque fois augmentée par une contrainte imposant que obj_1 doit être meilleur que celui de la solution trouvée. Nous détaillerons cet algorithme dans la section 4. Pour trouver toutes les solutions de Pareto, la recherche est donc lancée plusieurs fois. Une autre approche est proposée dans [11], qui considère une contrainte Pareto sous forme d'une contrainte globale. La contrainte Pareto maintient un ensemble $\mathcal{A}$ des solutions non dominées et opère sur les variables représentant les critères. Avec cette contrainte globale, la recherche est lancée une seule fois pour trouver toutes les solutions de Pareto. Un cadre MO-LNS intégrant l'optimisation multi-objectif avec la contrainte globale de Pareto et Large Neighborhood Search est présenté dans [17].

3 Modélisation des tâches de clustering sous contraintes utilisateur

Nous disposons d'une collection de n points et d'une mesure de dissimilarité d entre paires de points. Sans perte de généralité, nous supposons que les points sont indexés et nommés par leur indice (ainsi 1 représente le 1er point). Dans [5, 6], nous avons présenté un modèle en PPC pour la tâche imposant que le nombre k de classes soit fixé à l'avance par l'utilisateur. Dans cette section, nous présentons un nouveau modèle, qui est différent du modèle précédent dans le choix de variables et de contraintes. Ce modèle est plus flexible, car le nombre k n'est plus fixé mais borné entre k_{min} classes et k_{max} classes, où k_{min} et k_{max} sont des paramètres fixés par l'utilisateur. Ce modèle est aussi plus efficace en temps et en espace que le modèle précédent.

3.1 Variables

Les classes (clusters) sont identifiées par leur numéro, qui varie de 1 à k, pour une partition en k classes. Pour représenter l'affectation des points aux classes, nous utilisons des variables à valeurs entières $G[1], \ldots, G[n]$, dont le domaine est l'ensemble des entiers de $[1, k_{max}]$. Une affectation $G[i] = c$ signifie que le point i est affecté à la c-ème classe.

Une variable est aussi introduite pour représenter le critère à optimiser. Elle est écrite D pour le diamètre maximal, S pour la marge minimale. Elle est à valeur réelle, puisque les distances sont réelles. Les domaines de D et S sont des intervalles dont la borne inférieure (supérieure) est la dissimilarité minimale (resp. maximale) des paires de points.

3.2 Contraintes

La tâche de classification non supervisée sous contraintes utilisateur est modélisée par des contraintes de la Programmation par Contraintes.

3.2.1 Contraintes de partitionnement

Ces contraintes expriment les relations entre les points et leurs clusters. Dans le modèle précédent [5], chaque cluster est identifié par son représentant, qui est le point de plus petit indice du cluster. Dans ce nouveau modèle, les clusters ne sont pas identifiés par leur représentant, mais par leur numéro. Avec cette identification, une partition pourrait avoir différentes représentation, qui sont obtenues par des permutations des indices de l'ensemble de clusters. Une permutation des indices peut se produire lors de la création d'un nouveau cluster, si un numéro quelconque parmi les numéros restants lui est attribué.

Pour casser ces symétries de valeurs, les clusters seront numérotés de telle manière qu'un nouveau numéro c, avec $c > 1$, soit utilisé pour un nouveau cluster si et seulement si le numéro $c-1$ a déjà été utilisé. Cela peut être réalisé avec une contrainte *precede* [15] :

$$precede([G[1], \ldots, G[n]], [1..k_{max}]).$$

Cette contrainte impose que $G[1] = 1$, et de plus, si $G[i] = c$ avec $1 < c \leq k_{max}$, il existe au moins une variable $G[j]$ avec $j < i$ et $G[j] = c - 1$.

Que le nombre minimal de clusters soit k_{min} signifie que tous les numéros entre 1 et k_{min} doivent être utilisés dans les affectations de $G[i]$. Avec la contrainte *Precede*, il suffit d'exiger qu'il y ait au moins une variable $G[i]$ égale à k_{min}. Autrement dit, la valeur k_{min} doit être prise au moins une fois par des variables de $G[1], \ldots, G[n]$. Ceci s'exprime par une contrainte "count" :

$$\#\{i \mid G[i] = k_{min}\} \geq 1.$$

Si l'utilisateur souhaite exactement k classes, il suffit de fixer $k_{min} = k_{max} = k$.

3.2.2 Modélisation des contraintes utilisateur

Les contraintes utilisateur classiques peuvent être intégrées directement. Nous présentons les contraintes utilisateur nécessaires pour ce papier, pour les autres types de contraintes nous renvoyons le lecteur vers [5].

- Une δ-contrainte exprime que la marge entre deux clusters doit être au moins δ. Ceci est représenté par $S \geq \delta$. De plus, pour chaque couple (i, j), $i < j \in [1, n]$ satisfaisant $d(i, j) < \delta$, nous posons la contrainte : $G[i] = G[j]$.
- Une contrainte de diamètre exprime que le diamètre d'un cluster ne doit pas dépasser γ. Elle est représentée par $D \leq \gamma$. De plus, pour chaque couple (i, j), $i < j \in [1, n]$ tel que $d(i, j) > \gamma$, nous posons la contrainte : $G[i] \neq G[j]$.
- Une contrainte must-link entre deux points i et j est exprimée par : $G[i] = G[j]$.
- Une contrainte cannot-link sur i et j est exprimée par : $G[i] \neq G[j]$.

3.3 Critère du diamètre ou de la marge

L'optimisation d'un critère est réalisée par le mécanisme *branch-and-bound*. Chaque fois qu'une nouvelle solution est trouvée, de nouvelles contraintes sont ajoutées pour interdire des solutions moins bonnes. Lorsque le critère est de minimiser le diamètre maximal D, à chaque fois qu'une solution Δ est trouvée, la valeur du diamètre maximal $D(\Delta)$ est calculée. Cette valeur est utilisée pour poser une nouvelle contrainte $D < D(\Delta)$. De plus, pour chaque couple (i, j), $i < j \in [1, n]$ satisfaisant $d(i, j) \geq D(\Delta)$, nous posons la contrainte : $G[i] \neq G[j]$.

Lorsque le critère est de maximiser la marge minimale S entre clusters, chaque fois qu'une solution Δ est trouvée, la valeur de la marge $S(\Delta)$ est calculée et la contrainte $S > S(\Delta)$ est posée. De plus, pour chaque couple (i, j), $i < j \in [1, n]$ satisfaisant $d(i, j) \leq S(\Delta)$, nous posons la contrainte : $G[i] = G[j]$.

3.4 Stratégie de recherche

Pour améliorer l'efficacité, les points sont initialement réordonnés suivant la stratégie présentée dans notre précédent travail [5]. Le branchement est réalisé sur les variables de G. Lors de chaque choix, une variable $G[i]$ avec le plus petit domaine restant est choisie en premier. Quand $G[i]$ est choisie, toutes les valeurs c du domaine de $G[i]$ sont examinées et nous choisissons le numéro de classe la plus proche du point i. La distance entre un point i et une classe c est définie comme la distance maximale entre i et tous les points j pour lesquels $G[j]$ est instancié et $G[j] = c$. Si une classe c est vide (il n'existe pas de point instancié $G[j]$ tel que $G[j] = c$) la distance entre i et cette classe est nulle. Cela signifie que l'on privilégie d'affecter le point à une nouvelle classe s'il reste un numéro ; de plus, le premier numéro de la classe vide suffit. La classe c_0 la plus proche de i est choisie et le branchement sur $G[i]$ fait deux alternatives $G[i] = c_0$ et $G[i] \neq c_0$. Cette stratégie diffère de celle utilisée dans le premier modèle, où le branchement dépendait de la distance entre le point i et le représentant de chaque classe.

4 Clustering bi-critère marge-diamètre

On considère maintenant la tâche de clustering sous un ensemble de contraintes utilisateur $\mathcal{E}$ et le bi-critère marge-diamètre, avec $k \geq 2$. La déclarativité du modèle offre à l'utilisateur le choix du critère d'optimisation et la capacité d'intégrer différents types de contraintes utilisateur. Avec cette déclarativité, notre modèle peut s'utiliser directement pour trouver un ensemble complet et minimal d'optima de Pareto pour ce problème. Ceci est réalisé soit par des optimisations mono-critère successives utilisant des contraintes utilisateur, soit en utilisant la contrainte globale Pareto.

4.1 Optimisations mono-critère successives

L'algorithme de van Wassenhove et Gelders [24] permet de calculer l'ensemble des solutions de Pareto pour un problème bi-critère, par exemple $(\max obj_1, \max obj_2)$ par des optimisations mono-critère successives :

1. Chercher une solution Δ optimisant $\max obj_2$.
2. Si Δ n'existe pas, aller à l'étape 4. Sinon, Δ est ajoutée dans l'ensemble $\mathcal{A}$.
3. Imposer $obj_1 > obj_1(\Delta)$ et revenir à l'étape 1.
4. Supprimer les solutions dominées dans $\mathcal{A}$.

Avec la capacité qu'à notre modèle d'intégrer des contraintes utilisateur, il offre un cadre pour réaliser directement cette approche pour l'optimisation bi-critère $(\max S, \min D)$. Ceci est présenté dans l'algorithme 1. Nous désignons par *Minimise_Diamètre(CU)* ou *Maximise_Marge(CU)*, avec CU un ensemble de contraintes utilisateur, une fonction qui appelle notre modèle pour la minimisation du diamètre maximal (ou pour la maximisation de la marge minimale, resp.) sous les contraintes dans CU. Ces fonctions retournent une solution optimale Δ qui satisfait toutes les contraintes utilisateurs dans l'ensemble CU, si une telle solution existe et $NULL$ sinon. La fonction *Réduire*($\mathcal{A}$) supprime toutes les solutions dominées dans l'ensemble $\mathcal{A}$.

Algorithme 1 : Algorithme B1

```
1 A ← ∅ ;
2 Δ ← Minimise_Diamètre(E) ;
3 tant que Δ ≠ NULL faire
4 |   A ← A ∪ {Δ}
5 |   Sep ← S(Δ)
6 |   Δ ← Minimise_Diamètre(E ∪ {S > Sep}) ;
7 Réduire(A) ;
```

Une observation importante concernant l'algorithme B1 est qu'à la ligne 6, on peut rencontrer plusieurs solutions avec la marge améliorée petit à petit (grâce à la contrainte utilisateur $S > Sep$), mais toujours avec le même diamètre. Chacune de ces solutions n'est donc pas dominée à une étape mais devient dominée à l'étape suivante, lorsqu'une nouvelle solution avec le même diamètre mais avec une meilleure marge est trouvée. Ces solutions intermédiaires ne sont donc pas des solutions de Pareto, elles seront éliminées lors de la dernière étape. Pour éviter ces solutions non Pareto, nous pouvons chercher la marge optimale sous condition que le diamètre n'est pas pire. Les critères du diamètre et de la marge sont donc considérés successivement, comme présenté dans l'algorithme 2. Cet algorithme est similaire à la méthode ϵ-contrainte [18]. Nous montrons que l'ensemble $\mathcal{A}$ calculé est un ensemble complet et minimal d'optima de Pareto.

Algorithme 2 : Algorithme B2

```
1 A ← ∅ ;
2 i ← 1 ;
3 Δ_i^D ← Minimise_Diamètre(E) ;
4 tant que Δ_i^D ≠ NULL faire
5 |   Δ_i^S ← Maximise_Marge(E ∪ {D ≤ D(Δ_i^D)}) ;
6 |   A ← A ∪ {Δ_i^S} ;
7 |   i ← i + 1 ;
8 |   Δ_i^D ← Minimise_Diamètre(E ∪ {S > S(Δ_{i-1}^S)}) ;
```

Proposition 4.1. *Soient $\Delta_1^D, \Delta_1^S, \ldots, \Delta_m^D, \Delta_m^S$ les solutions visitées par l'algorithme B2. On a :*

1. *si $\Delta_i^D \neq NULL$ alors $\Delta_i^S \neq NULL$,*
2. *pour tout $1 < i \leq m$, $S(\Delta_i^S) > S(\Delta_{i-1}^S)$,*
3. *pour tout $1 < i \leq m$, $D(\Delta_i^D) > D(\Delta_{i-1}^D)$,*
4. *pour tout $1 \leq i \leq m$, $D(\Delta_i^S) = D(\Delta_i^D)$,*
5. *$\nexists \Delta$ tel que $D(\Delta) < D(\Delta_1^D)$,*
6. *pour tout $1 \leq i < m$, $\nexists \Delta$ tel que $S(\Delta) \geq S(\Delta_i^S)$ et $D(\Delta) < D(\Delta_{i+1}^D)$.*

Preuve

1. Si $\Delta_i^D \neq NULL$, l'ensemble des partitions satisfaisant $\mathcal{E} \cup \{D \leq D(\Delta_i^D)\}$ contient au moins Δ_i^D, donc au moins $\Delta_i^S = \Delta_i^D$.
2. Puisque Δ_i^D et Δ_i^S satisfont la condition $D \leq D(\Delta_i^D)$, et que Δ_i^S est celle qui maximise la marge (ligne 5), on a $S(\Delta_i^S) \geq S(\Delta_i^D)$. Puisque Δ_i^D satisfait la condition $S > S(\Delta_{i-1}^S)$ (ligne 8), on a $S(\Delta_i^D) > S(\Delta_{i-1}^S)$. Donc $S(\Delta_i^S) > S(\Delta_{i-1}^S)$.
3. L'ensemble des partitions satisfaisant $S > S(\Delta_i^S)$ est un sous-ensemble strict de l'ensemble des partitions satisfaisant $S > S(\Delta_{i-1}^S)$, car $S(\Delta_i^S) > S(\Delta_{i-1}^S)$. Puisque Δ_i^D et Δ_{i-1}^D sont des partitions qui minimisent le diamètre parmi les éléments de ces deux ensembles respectifs, on a $D(\Delta_i^D) \geq D(\Delta_{i-1}^D)$. Si $D(\Delta_i^D) = D(\Delta_{i-1}^D)$, par la ligne 5 on a $S(\Delta_i^S) = S(\Delta_{i-1}^S)$ ce qui contredit $S(\Delta_i^S) > S(\Delta_{i-1}^S)$. On a donc $D(\Delta_i^D) > D(\Delta_{i-1}^D)$.
4. Par la ligne 5, on a $D(\Delta_i^S) \leq D(\Delta_i^D)$. Puisque $S(\Delta_i^S) > S(\Delta_{i-1}^S)$ et Δ_i^D est une partition qui minimise le diamètre parmi les partitions satisfaisant la condition $S > S(\Delta_{i-1}^S)$ (ligne 8), on a $D(\Delta_i^S) \geq D(\Delta_i^D)$. Donc $D(\Delta_i^S) = D(\Delta_i^D)$.
5. Par la ligne 3, Δ_1^D est une partition ayant le diamètre minimal parmi toutes celles qui satisfont $\mathcal{E}$, alors il n'existe pas de partition Δ satisfaisant $\mathcal{E}$ telle que $D(\Delta) < D(\Delta_1^D)$.
6. Supposons qu'il y ait une solution Δ telle que $S(\Delta) \geq S(\Delta_i^S)$ et $D(\Delta) < D(\Delta_{i+1}^D)$. Puisque $S(\Delta) \geq S(\Delta_i^S)$ et Δ_{i+1}^D est la partition qui minimise le diamètre parmi celles qui satisfont la contrainte $S > S(\Delta_i^S)$, on a $D(\Delta) \geq D(\Delta_{i+1}^D)$. Ceci contredit le fait que $D(\Delta) < D(\Delta_{i+1}^D)$. □

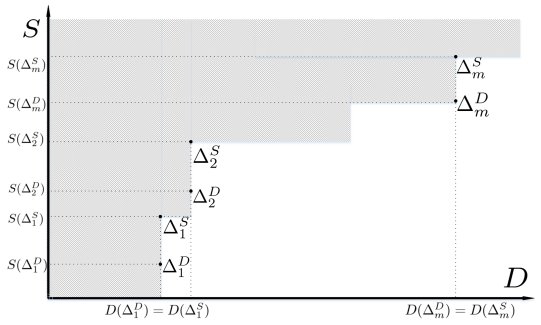


FIGURE 2 – Les solutions trouvées par l'algorithme B2

Le positionnement des solutions trouvées par l'algorithme B2 est présenté dans la figure 2. Il n'existe pas de solution dans la zone grisée. Une solution dans la zone blanche est dominée par une solution $\Delta_i^S \in \mathcal{A}$.

Proposition 4.2. *L'ensemble* $\mathcal{A} = \{\Delta_1^S, \ldots, \Delta_m^S\}$ *calculé par l'algorithme B2 est complet et minimal.*

Preuve L'algorithme s'arrête car d'après la proposition 4.1, $S(\Delta_i^S) > S(\Delta_{i-1}^S)$ et ces valeurs sont discrètes et limitées par la distance maximale entre deux points.

On montre que chaque Δ_i^S $(1 \leq i \leq m)$ est une solution de Pareto, *i.e.* il n'existe pas de partition Δ telle que soit $D(\Delta) \leq D(\Delta_i^S)$ et $S(\Delta) > S(\Delta_i^S)$, soit $D(\Delta) < D(\Delta_i^S)$ et $S(\Delta) \geq S(\Delta_i^S)$. Puisque $D(\Delta_i^S) = D(\Delta_i^D)$, la partition Δ doit satisfaire soit $D(\Delta) \leq D(\Delta_i^D)$ et $S(\Delta) > S(\Delta_i^S)$, soit $D(\Delta) < D(\Delta_i^D)$ et $S(\Delta) \geq S(\Delta_i^S)$. Le premier cas est impossible car Δ_i^S est une partition qui maximise la marge parmi celles satisfaisant la condition $D \leq D(\Delta_i^D)$. Pour le second cas, si $i = 1$ alors Δ n'existe pas d'après le point 5 de la proposition 4.1. Si $i > 1$, puisque $S(\Delta_i^S) > S(\Delta_{i-1}^S)$, la partition Δ doit satisfaire $D(\Delta) < D(\Delta_i^D)$ et $S(\Delta) > S(\Delta_{i-1}^S)$. Ceci est impossible d'après le point 6 de la proposition 4.1.

Soit Δ une solution de Pareto telle que $\Delta \notin \mathcal{A}$. Deux cas peuvent se produire : soit il existe $\Delta_i^S \in \mathcal{A}$ telle que Δ est équivalente à Δ_i^S , soit une telle Δ_i^S n'existe pas. Dans le deuxième cas, d'après les points 5 et 6 de la proposition 4.1, Δ pour ne pas être dominée doit avoir $S(\Delta) > S(\Delta_m^S)$. Or l'algorithme s'arrête après Δ_m^S, *i.e.* il n'existe pas de partition ayant une marge plus grande que $S(\Delta_m^S)$. Donc le deuxième cas ne peut exister. D'autre part, d'après le point 2 de la proposition 4.1, dans $\mathcal{A}$ il n'existe pas deux partitions qui sont équivalentes. L'ensemble $\mathcal{A}$ est donc complet et minimal. □

4.2 Utilisation de la contrainte Pareto

Une optimisation multi-objectif en Programmation par Contraintes en une seule phase de recherche est proposée dans [11]. L'idée est de réaliser une contrainte globale $Pareto(Obj_1, \ldots, Obj_m, \mathcal{A})$, qui maintient un ensemble de solutions non dominées $\mathcal{A}$ et qui opère sur les variables de fonction objectif Obj_i. Cette contrainte a été améliorée et une extension avec Large Neighborhood Search est proposée dans [17].

La contrainte *Pareto* peut être introduite dans notre modèle pour le problème de classification non supervisée avec le bi-critère marge-diamètre. Le filtrage de cette contrainte [17] appliqué dans notre cas peut se résumer dans l'algorithme 3, où $D \in [D.min, D.max)$ et $S \in (S.min, S.max]$. Ici $D.max$ et $S.min$ sont des bornes strictes de D et S. Pour que cette contrainte soit utile, nous ajoutons des contraintes pour relier le diamètre, la marge avec les points.

– Deux points à une distance supérieure au diamètre maximal doivent être dans des clusters différents. Ceci est représenté par les contraintes réifiées suivantes [1] : $\forall i, j \in [1, n]$, $i < j$,

$$d(i, j) > D \rightarrow G[i] \neq G[j]. \quad (1)$$

– Deux points à une distance inférieure à la marge minimale doivent être dans le même cluster : $\forall i, j \in [1, n]$, $i < j$,

$$d(i, j) < S \rightarrow G[i] = G[j]. \quad (2)$$

Algorithme 3 : Filtrage pour la contrainte *Pareto*

```
pour chaque solution Δ_i ∈ A faire
    si D.min ≥ D(Δ_i) alors
        S.min ← max(S.min, S(Δ_i))
    si S.max ≤ S(Δ_i) alors
        D.max ← min(D.max, D(Δ_i))
```

L'avantage de cette approche est que toutes les solutions optimales de Pareto sont trouvées dans une seule phase de recherche. Néanmoins, le filtrage de la contrainte *Pareto*, guidé par les solutions dans l'ensemble $\mathcal{A}$, n'est efficace que lorsque les éléments de $\mathcal{A}$ sont près des solutions de Pareto. Ceci nécessite des études plus raffinées par exemple sur la stratégie de recherche pour avoir un meilleur ensemble $\mathcal{A}$. De plus, le modèle est plus lourd que celui présenté dans la section 3 car il faut utiliser plusieurs contraintes réifiées.

1. Une contrainte réifiée $A \rightarrow B$ représente la relation implication, la contrainte B devient effective si la contrainte A est satisfaite, et si la contrainte B n'est pas satisfaite alors $\neg A$ doit être satisfaite.

Bases de données	# Objets	# Classes
Iris	150	3
Wine	178	3
Glass	214	7
Ionosphere	351	2
User Knowledge	403	4
WDBC	569	2
Synthetic Control	600	6
Vehicle	846	4
Yeast	1484	10
Wine Quality	1599	4
Image Segmentation	2000	7

TABLE 1 – Propriétés des bases de données

Bases de données	D_{opt}	CP	CdG	BaB
Iris	2.58	< 0.1	1.8	1.4
Wine	458.13	< 0.1	2.3	2
Glass	4.97	< 0.1	8.1	42
Ionosphere	8.6	0.2	0.6	-
User Knowledge	1.17	0.2	3.7	-
WDBC	2377.96	0.5	1.8	-
Synthetic Control	109.36	1.4	-	-
Vehicle	264.83	0.9	-	-
Yeast	0.67	4.9	-	-
Wine Quality	69.78	3.1	-	-
Image Segmentation	436.4	5.5	-	-

TABLE 2 – Performances avec la minimisation du diamètre maximal

5 Expérimentations

Nous avons implanté notre modèle en utilisant la bibliothèque de programmation par contraintes Gecode version 4.2.1 [2]. Onze bases de données du répertoire UCI [1] sont utilisées dans nos expériences. Elles varient en fonction de leur taille et du nombre de clusters. Pour la base *Wine Quality*, le nombre de classes est inconnu et nous choisissons $k = 4$ pour les tests. Le tableau 1 résume les informations sur ces bases, qui sont présentées suivant le nombre croissant d'objets. Les expérimentations sont réalisées sur un processeur 3.4GHz Core i5 Intel sous Ubuntu 12.04.

5.1 Clustering avec le critère de minimiser le diamètre maximal

Nous comparons notre modèle (noté CP) avec l'algorithme basé sur la recherche *branch-and-bound* [3] (noté BaB) [3] et l'algorithme basé sur la coloration de graphe [10] (noté CdG) [4]. Nous considérons le cas sans contraintes utilisateur car les algorithmes auxquels nous nous comparons ne peuvent pas traiter de contraintes et à notre connaissance, il n'existe pas d'algorithme exact avec des contraintes utilisateur pour ce critère. Dans les expérimentations, le temps est limité à 1 heure et la distance euclidienne est utilisée pour calculer la dissimilarité entre objets. Le nombre de classes k est fixé au nombre de classes réel donné dans le tableau 1 pour les 3 algorithmes (pour notre modèle, $k_{min} = k_{max} = k$).

Le tableau 2 montre les résultats des expérimentations. Pour chaque base nous présentons la valeur D_{opt} du diamètre optimal trouvé ainsi que le temps d'exécution en secondes de chaque système. Le signe - est utilisé lorsque le système ne trouve pas la solution optimale après 1 heure. Tous les systèmes trouvent naturellement le même diamètre optimal.

Les résultats montrent que notre modèle a la meilleure perfomance dans tous les cas. Parmi les 3 programmes, l'algorithme BaB est le moins efficace, n'étant pas capable de résoudre les bases de données de plus de 300 objets. La performance de CdG est rapidement réduite si le nombre d'objets passe à 500. Ces algorithmes n'ont pas de mécanisme de réduction de domaines de variables. L'algorithme BaB se base sur les bornes pour détecter les échecs pendant la recherche, tandis que CdG examine toutes les distances disponibles dans l'ordre décroissant pour trouver le diamètre optimal. Notre modèle, qui exploite les avantages de la Programmation par Contraintes, comme la propagation de contraintes ou des stratégies de recherche adaptées, est plus performant.

Notre nouveau modèle montre une forte amélioration de l'efficacité par comparaison avec l'ancien modèle dont les résultats sont présentés dans [5]. Par exemple pour la base Synthetic Control le temps avec le nouveau modèle est de $1,4$s vs. $56,1$s pour l'ancien modèle, pour la base Vehicle le temps est de $0,9$s vs. $14,9$s, et pour la base Yeast nous avons $4,9$s vs. $2389,9$s.

5.2 Clustering avec bi-critère marge-diamètre

Pour le bi-critère marge-diamètre, nous utilisons notre modèle avec les algorithmes B1, B2 et l'utilisation de la contrainte Pareto (noté P). Nous comparons avec l'algorithme proposé dans [10] (noté H) [5]. A notre connaissance, c'est le seul algorithme exact pour le cas où le nombre de classes k est supérieur à 2.

Le premier test est sur l'exemple donné dans la figure 1, avec le nombre de classes fixé à 3. Dans le

2. http ://www.gecode.org

3. Le programme peut être trouvé à l'adresse http ://mailer.fsu.edu/~mbrusco/

4. Le programme est codé par nous-même en C++.

5. Le programme est codé par nous-même en C++.

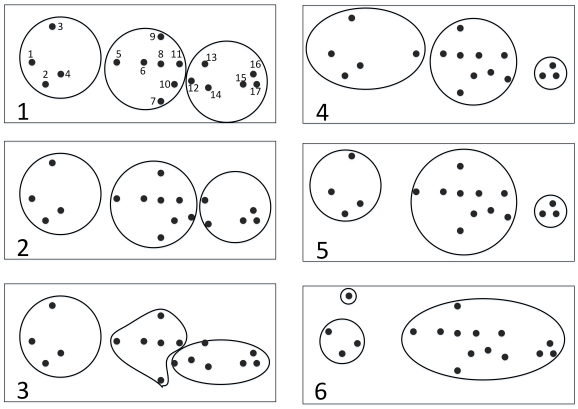


FIGURE 3 – Les solutions de Pareto

Contraintes utilisateur	Solutions trouvées
Must-link(5, 14)	5 et 6
Must-link(5, 14) et taille de groupe ≥ 2	5
Cannot-link(1, 5)	1,2, 3,5 et 6
$S \geq 0.1d(1, 17)$	4, 5 et 6
Cannot-link(1, 5) et $S \geq 0.1d(1, 17)$	5 et 6

TABLE 3 – Les solutions avec contraintes utilisateur

cas sans contraintes utilisateur, les 4 programmes obtiennent le même résultat. Les solutions sont détaillées dans la figure 3. Si on maximise le ratio entre la marge et le diamètre, la solution 5 est la meilleure avec la valeur de ratio égale à 0.36. C'est aussi la solution qui semble la plus raisonnable.

Notre modèle offre en plus la possibilité d'intégrer des contraintes utilisateur. Pour atteindre des solutions souhaitées, l'utilisateur peut combiner différents types de contraintes. Le tableau 3 montre les solutions obtenues avec différentes combinaisons de contraintes utilisateur.

Nous comparons la performance des algorithmes en utilisant les bases de données du tableau 1. Le temps est limité à 1 heure. Le nombre de classes k varie entre 2 et le nombre de classes réel. Le tableau 4 présente le temps d'exécution de chaque approche, en secondes. La deuxième colonne (#S) indique le nombre de solutions de Pareto trouvées.

Parmi les 4 programmes, l'approche P est la moins efficace. Bien que la contrainte *Pareto* nous permette de trouver toutes les solutions en une seule phase de recherche, il manque des techniques pour bien réduire l'espace de recherche. Ceci n'est pas dû à la gestion interne de l'ensemble $\mathcal{A}$ dans la contrainte, car le nombre

Base de données	#S	B1	B2	P	H
Iris	8	<0.1	<0.1	7.8	4.2
Wine	3	0.4	0.2	-	0.9
Glass	9	0.4	0.2	-	21.5
Ionosphere	6	4.8	1.1	-	1.8
User Knowledge	16	8.9	2.2	-	23.6
WDBC	7	0.9	0.9	-	167.5
Synthetic Ctrl	6	49.6	3.1	-	-
Vehicle	13	3.6	3.8	-	-
Yeast	-	-	-	-	-
Wine Quality	12	30.9	22.7	-	-
Image Seg	8	65.8	41.3	-	-

TABLE 4 – Performance des approches de clustering bi-critère

de ces solutions, comme indiqué dans la colonne #S, est assez faible. De plus, la propagation des contraintes réifiées ajoutées dans le modèle est plus coûteuse que les contraintes simples (must-link ou cannot-link). On peut constater que l'algorithme B2 a la meilleure performance dans tous les cas. Il profite de la propagation efficace des contraintes ajoutées à chaque itération pour réduire l'espace de recherche. De plus, en optimisant aussi la marge, le modèle dans l'algorithme B2 est relancé moins souvent que dans l'algorithme B1. Comme CdG, l'algorithme H est limité avec les bases de moins de 500 points.

6 Conclusion

Dans ce papier, nous présentons un nouveau modèle en Programmation par Contraintes pour les tâches de classification non supervisée sous contraintes utilisateur. Dans ce modèle sont intégrés différents types de contraintes utilisateur et les critères de minimisation du diamètre maximal des clusters D ou de maximisation de la marge minimale entre clusters S. Nous montrons que la généricité et la déclarativité du modèle permettent de l'utiliser directement pour traiter le clustering bi-critère ($\max S, \min D$) et sous contraintes utilisateur. Des expérimentations sur des bases de données classiques montrent que l'utilisation de notre modèle est plus efficace que les algorithmes exacts existants.

Pour le moment l'approche avec la contrainte globale *Pareto* n'est pas encore très compétitive. Afin de rendre plus efficace l'utilisation de cette contrainte, nous envisageons d'étudier des stratégies de recherche adaptées, éventuellement étendues avec LNS. D'autre part, notre modèle peut s'utiliser pour d'autres types de bi-critère, lorsqu'il permet de représenter chaque critère individuellement. Nous envisageons d'étendre

le modèle afin que d'autres critères puissent s'intégrer, par exemple le critère populaire de minimisation de la somme intra-cluster des distances au carré.

Références

[1] K. Bache and M. Lichman. UCI Machine Learning Repository, 2013.

[2] M. Bilenko, S. Basu, and R. J. Mooney. Integrating constraints and metric learning in semi-supervised clustering. In *Proceedings of the Twenty-First International Conference on Machine Learning*, pages 11–18, 2004.

[3] Michael Brusco and Stephanie Stahl. *Branch-and-Bound Applications in Combinatorial Data Analysis (Statistics and Computing)*. Springer, 1 edition, July 2005.

[4] R. Cormack. A review of classification. *Journal of the Royal Statistical Society. Series A (General)*, 134(3) :321–367, 1971.

[5] Thi-Bich-Hanh Dao, Khanh-Chuong Duong, and Christel Vrain. A Declarative Framework for Constrained Clustering. In *European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases ECML/PKDD*, 2013.

[6] Thi-Bich-Hanh Dao, Khanh-Chuong Duong, and Christel Vrain. Un modèle général pour la classification non supervisée sous contraintes utilisateur. In *Neuvième Journées Francophones de Programmation par Contraintes*, 2013.

[7] I. Davidson and S. S. Ravi. Agglomerative hierarchical clustering with constraints : Theoretical and empirical results. In *Proceedings of the 9th European Conf. on Principles and Practice of Knowledge Discovery in Databases*, pages 59–70, 2005.

[8] Ian Davidson and S. S. Ravi. Clustering with Constraints : Feasibility Issues and the k-Means Algorithm. In *Proc. 5th SIAM Data Mining Conference*, 2005.

[9] Ian Davidson and S. S. Ravi. The Complexity of Non-hierarchical Clustering with Instance and Cluster Level Constraints. *Data Mining Knowledge Discovery*, 14(1) :25–61, 2007.

[10] Michel Delattre and Pierre Hansen. Bicriterion Cluster Analysis. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-2(4) :277–291, 1980.

[11] Marco Gavanelli. An algorithm for multi-criteria optimization in csps. In Frank van Harmelen, editor, *ECAI*, pages 136–140. IOS Press, 2002.

[12] T. Gonzalez. Clustering to minimize the maximum intercluster distance. *Theoretical Computer Science*, 38 :293–306, 1985.

[13] Pierre Hansen and Michel Delattre. Complete-link cluster analysis by graph coloring. *Journal of the American Statistical Association*, 73(362) :397–403, 1978.

[14] StephenC. Johnson. Hierarchical clustering schemes. *Psychometrika*, 32(3) :241–254, 1967.

[15] Yat Chiu Law and Jimmy Ho-Man Lee. Global constraints for integer and set value precedence. In Mark Wallace, editor, *CP*, volume 3258 of *Lecture Notes in Computer Science*, pages 362–376. Springer, 2004.

[16] Zhengdong Lu and Miguel A. Carreira-Perpinan. Constrained spectral clustering through affinity propagation. In *2008 IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–8, 2008.

[17] Pierre Schaus and Renaud Hartert. Multi-Objective Large Neighborhood Search. In *Principles and Practice of Constraint Programming CP*, pages 611–627, 2013.

[18] Vincent T'kindt and Jean-Charles Billaut. *Multicriteria Scheduling, Theory, Models and Algorithms*. Springer, 2 edition, 2006.

[19] K. Wagstaff and C. Cardie. Clustering with instance-level constraints. In *Proceedings of the Seventeenth International Conference on Machine Learning*, pages 1103–1110, 2000.

[20] K. Wagstaff, C. Cardie, S. Rogers, and S. Schroedl. Constrained k-means clustering with background knowledge. In *Proceedings of the 18th International Conference on Machine Learning*, pages 577–584, 2001.

[21] Jiabing Wang and Jiaye Chen. Clustering to maximize the ratio of split to diameter. In *ICML*, 2012.

[22] Xiang Wang and Ian Davidson. Flexible constrained spectral clustering. In *Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 563–572, 2010.

[23] Y. Wang, H. Yan, and C. Sriskandarajah. The weighted sum of split and diameter clustering. *Journal of Classification*, 13(2) :231–248, 1996.

[24] Luc N. Van Wassenhove and Ludo F. Gelders. Solving a bicriterion scheduling problem. *European Journal of Operational Research*, 4(1) :42 – 48, 1980.

Parallélisation Portfolio de Solveur PPC

Tarek Menouer [1] **Bertrand Le Cun**[1]

[1] Laboratoire PRISM, 45 avenue des Etats-Unis 78035 Versailles
{Tarek.menouer,Bertrand.lecun}@prism.uvsq.fr

Résumé

Cet article présente un solveur de Programmation Par Contraintes (PPC) parallèle, basé sur la méthode du Portfolio pour résoudre rapidement les problèmes de satisfaction et d'optimisation de contraintes. Le Portfolio est largement utilisé dans la parallélisation des solveurs de SATisfiabilité booléenne (SAT) et les solveurs de PPC. Le principe consiste à exécuter plusieurs stratégies de recherche pour résoudre le même problème, en utilisant différents cœurs de calcul. Classiquement une stratégie de recherche est exécutée sur un cœur de calcul (Portfolio N To N). La première stratégie qui répond aux besoins de l'utilisateur arrête toutes les autres stratégies. En utilisant une parallélisation interne pour chaque stratégie on peut exécuter N stratégies sur P cœurs de calculs avec $P > N$ (Portfolio N To P). La nouveauté, consiste à adapter l'ordonnancement des N stratégies de recherche entre elles afin de privilégier la stratégie la plus prometteuse et de lui donner plus de cœurs que les autres. Les performances obtenues en utilisant le solveur Portfolio sont illustrées par la résolution des problèmes de PPC modélisés en utilisant le format FlatZinc et résolus avec le solveur Google OR-Tools au-dessus de notre framework parallèle Bobpp.

1 Introduction

Nous présentons dans cet article un Portfolio Adaptatif pour résoudre les problèmes de Programmation Par Contraintes (PPC). La résolution des problèmes de PPC est basée sur le choix des stratégies de recherche. Une stratégie de recherche est un algorithme qui choisit pour chaque nœud dans l'arbre de recherche, une variable à assigner avec une certaine valeur. Le problème est qu'on ne peut pas connaître à l'avance quelle est la meilleure stratégie pour résoudre un problème de PPC. Il existe plusieurs stratégies de recherche mais il n'existe pas une stratégie optimale pour résoudre tous les problèmes de PPC. D'un autre coté, les ressources de calcul sont de plus en plus puissantes et disponibles (les data-centers, le cloud computing, etc). Pour bénéficier de la variété des stratégies de recherche et de la puissance de calcul, nous présentons une méthode de parallélisation Portfolio. Le principe de Portfolio consiste à exécuter plusieurs stratégies de recherche sur différents cœurs de calcul. La première stratégie qui répond aux besoins de l'utilisateur, arrête toutes les autres stratégies. Le but de Portfolio est de résoudre tous les problèmes le plus rapidement possible, mais l'inconvénient est qu'on utilise beaucoup de ressources de calcul. Le Porfolio est utilisé principalement pour résoudre les problèmes de SATisfiabilité booléenne (SAT) [18, 16, 8]. Il existe plusieurs modèles de Portfolio. Le premier modèle et le Portfolio N To N, il consiste à exécuter N stratégies de recherche séquentielle sur P cœur de calcul. L'intérêt de Portfolio N To N est que la performance obtenue est celle de la meilleure stratégie, mais la faiblesse est que le nombre de stratégie est limité par comparaison avec le grand nombre de cœurs de calcul utilisés par les machines parallèles. En utilisant une parallélisation interne pour chaque stratégie de recherche, on peut exécuter N stratégies sur P cœurs ($P > N$), ce modèle de Portfolio est appelé le Portfolio N To P. La parallélisation interne consiste à partitionner l'arbre de recherche de chaque stratégie en un ensemble de sous-arbres, ensuite affecter ces sous-arbres aux différents cœurs de calcul [9]. La première manière de réaliser ce modèle de Portfolio est de partitionner l'arbre de recherche de chaque stratégie en $\frac{P}{N}$ sous-arbres, et ensuite affecter chaque sous-arbre à un cœur de calcul. La deuxième manière, consiste à affecter dynamiquement les P cœurs de calcul à la totalité des stratégies de recherche (les N stratégies), en utilisant la technique du vol de travail. Les deux modèles de Portfolio (N To N et N To P) gaspillent beaucoup de ressource de calcul car on ne sait pas décider à priori qu'elle est la meilleure stratégie, mais pendant la recherche on peut estimer l'avancement respectif des stratégies. Comme

toutes les stratégies sont ordonnancées par le même framework parallèle (Bobpp [2]), donc on peut contrôler l'ordonnancement des N stratégies entre elles et privilégier la stratégie la plus prometteuse, celle qu'on estime avoir l'arbre de recherche le plus petit, et on lui donne plus de cœurs que les autres stratégies. Ce modèle de Portfolio est appelé le Portfolio Adaptatif.

La section suivante détaille plus précisément la résolution des problèmes de PPC. Dans la section 3, les différents modèles de Portfolio sont présentés. Des expérimentations pour résoudre des problèmes de PPC modélisés en utilisant le format Flatzinc sont données dans la section 4. Enfin, une conclusion et quelques perspectives sont proposées en section 5.

2 La Résolution des Problèmes de Programation Par Contraintes

Un problème de PPC est constitué d'un ensemble de variables, $X = \{x_1, x_2, ..., x_n\}$, pour chaque variable $x \in X$, il existe un ensemble fini de domaines de valeurs, $D(x) = \{a_1, a_2, ..., a_k\}$ et une collection finie de contraintes, $C = \{c_1, c_2, ..., c_m\}$.

Un problème (γ) de PPC peut être résolu comme suit : Au début, toutes les variables de γ sont non assignées. A chaque étape, une variable x est choisie, et une valeur possible $a \in D(x)$ est assignée à son tour. Chaque branche d'un arbre de recherche calculée par cette recherche définit une assignation. Ensuite, le mécanisme de propagation vérifie, pour chaque valeur, la cohérence de cette assignation partielle avec les contraintes. En cas de cohérence, un appel récursif est effectué. Chaque assignation partielle crée un nœud dans l'arbre de recherche. Ainsi, nous associons la consistance d'un nœud avec la cohérence implicite d'une assignation.

Il existe plusieurs heuristiques pour choisir les variables de branchement, et pour chaque variable la valeur à assigner, ces heuristiques sont appelées les stratégies de recherche [14, 1, 6]. Les stratégies de recherche utilisées par le solveur OR-Tools [13] dépendent uniquement des données locales d'un nœud, c'est à dire, le branchement ne dépend pas de l'historique de la recherche.

OR-Tools est un solveur de PPC open source séquentiel, il est développé en C++ par l'équipe de recherche Google. Le principe de ce solveur est d'explorer des espaces de recherche pour trouver une ou toutes les solutions possibles en utilisant un algorithme de recherche en profondeur d'abord (Depth First Search *DFS* [1])

Par exemple, pour résoudre le problème $X/Y = Y$ avec,

— Le domaine des valeurs de la variable X = $\{9, 18\}$

— Le domaine des valeurs de la variable Y = $\{3, 6\}$

On peut choisir deux stratégies de recherche, parmi les différentes stratégies qui existent :

— **Stratégie Min Bound** : cette stratégie assigne pour chaque variable non assignée la plus petite valeur dans le domaine des valeurs. La figure 1 présente le déroulement de cette stratégie pour résoudre le problème $X/Y = Y$. Nous commençons par assigner à la variable X la plus petite valeur dans le domaine des valeurs, qui est la valeur 9. Ensuite, nous assignons à la variable Y la plus petite valeur qui est la valeur 3. Maintenant toutes les variables sont assignées. Nous testons si la contrainte $X/Y = Y$ est satisfaite, dans ce cas, $9/3 = 3$ est satisfait, donc une solution est trouvée en explorant 2 nœuds.

— **Stratégie Max Bound** : cette stratégie est l'inverse de la stratégie Min Bound, elle sélectionne pour chaque variable non assignée la plus grande valeur dans le domaine des valeurs. La figure 2 présente le déroulement de cette stratégie pour résoudre le problème $X/Y = Y$. La solution est trouvée en explorant 6 nœuds.

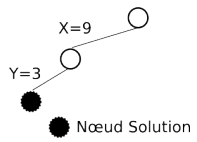


FIGURE 1 – Stratégie Min Bound

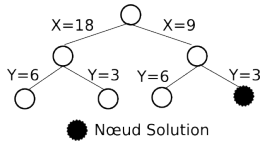


FIGURE 2 – Stratégie Max Bound

Comme on peut remarquer, pour résoudre un problème de PPC, la taille de l'arbre de recherche généré peut varier en fonction de la stratégie utilisée. Cette variation de taille influence fortement les performances obtenues.

Dans la section suivante, nous présentons quelques modèles de Portfolio pour résoudre les problèmes de PPC en utilisant le solveur OR-Tools au-dessus du framework parallèle Bobpp.

Bobpp est un framework parallèle open source, développé en C++ pour résoudre les problèmes d'optimisation combinatoire. Il peut être utilisé comme un support d'exécution. Le but de ce framework est de fournir une interface entre les solveurs de problèmes combinatoires et les machines parallèles en utilisant plusieurs environnements de programmation parallèle, tels que les POSIX threads, MPI, Hybride (POSIX threads+MPI) ou des bibliothèques plus spécialisées telles que Athapascasn/Kaapi [5].

3 Les Modèles de Portfolio

3.1 Portfolio N To N (N Stratégies $\times$ N Cœurs)

Le Portfolio N To N consiste à exécuter N stratégies de recherche sur N cœurs de calcul. Ensuite, la première stratégie qui répond aux besoins de l'utilisateur arrête toutes les autres stratégies, comme présenté dans la figure 3. Ce modèle de Portfolio est largement utilisé dans la parallélisation des solveurs SAT [18, 16, 8] et des solveurs de PPC [3].

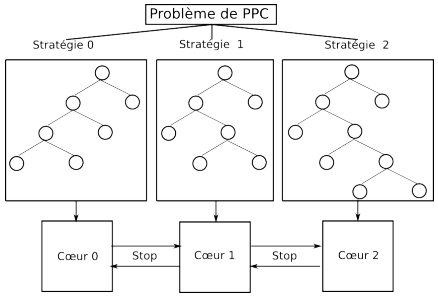


FIGURE 3 – Portfolio N To N

L'avantage de ce modèle de Portfolio est que la performance obtenue est celle de la meilleure stratégie [17], sauf que le nombre de stratégies de recherche qui existent dans la littérature est limité par comparaison avec le grand nombre de cœurs de calcul utilisés par les machines parallèles. Pour bénéficier de ce grand nombre de cœurs de calcul, nous proposons le Portfolio N To P avec $P > N$.

3.2 Portfolio N To P (N Stratégies $\times$ P Cœurs)

En utilisant une parallélisation interne pour chaque stratégie de recherche, on peut exécuter N stratégies de recherche sur P cœurs de calcul avec un bon équilibrage de charge entre les différents cœurs. Il existe plusieurs parallélisations internes [15, 7, 10, 4], dans notre cas nous avons choisi une parallélisation interne qui est présentée en détaille dans [9]. Elle consiste à partitionner l'arbre de recherche de chaque stratégie en un ensemble des sous-arbres disjoints, ensuite chaque sous-arbre est affecté à un cœur de calcul.

Pour réaliser ce modèle de Portfolio, il existe deux méthodes pour affecter les P cœurs de calcul au N stratégies de recherche : affectation statique ou dynamique.

3.2.1 Affectation Statique

L'affectation statique consiste à partitionner l'arbre de recherche de chaque stratégie en $\frac{P}{N}$ sous-arbres, et ensuite affecter statiquement chaque sous-arbre à un cœur de calcul. Généralement, les arbres de recherche générés pour résoudre les problèmes de PPC sont déséquilibrés, donc l'utilisation de cette affectation statique implique que parfois un cœur de calcul effectue quasiment la totalité de la recherche, tandis que les autres cœurs attendent que celui-ci termine sa recherche. Ce partitionnement donne un mauvais équilibrage de charge entre les cœurs de calcul, tel que présenté dans la figure 4. Pour résoudre ce problème, il existe une autre méthode d'affectation, qui est l'affectation dynamique.

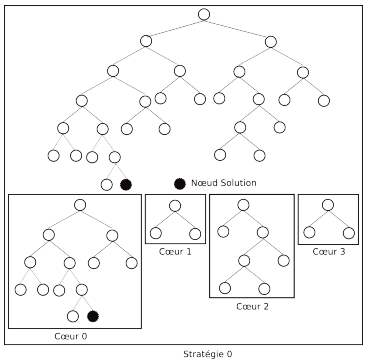


FIGURE 4 – L'équilibrage de charge pour une seule stratégie de recherche en utilisant l'affectation statique

3.2.2 Affectation Dynamique

Le principe de l'affectation dynamique est que l'arbre de recherche de chaque stratégie est partitionné entre les cœurs de calcul en demande et pendant l'exécution de l'algorithme de recherche, pour avoir un bon équilibrage de charge entre les cœurs. Pour partager les sous-arbres entre les cœurs de calcul, l'affectation dynamique utilise une File de Priorité Globale (FPG) implémentée dans le framework parallèle Bobpp. Chaque cœur de calcul effectue la recherche localement et de façon séquentielle en utilisant le solveur OR-Tools. La FPG contient initialement un nœud départ qui sera pris par le premier cœur de calcul pour commencer la recherche. Quand un cœur de calcul a terminé sa recherche dans son sous-arbre, il demande un nouveau sous-arbre à partir de la FPG. Si la FPG est vide, le cœur de calcul se déclare comme un cœur en attente. Les autres cœurs, les cœurs actifs, qui sont en train d'effectuer une recherche sur leurs sous-arbres, testent régulièrement s'il y a un cœur en attente. Si c'est le cas, le premier cœur actif qui détecte qu'il existe un cœur en attente, partitionne son sous-arbre en deux parties, il garde pour lui la partie droite et il insère la partie gauche dans la FPG. Le cœur en attente prend effet par l'insertion d'un nouveaux sous-arbre dans la

FPG.

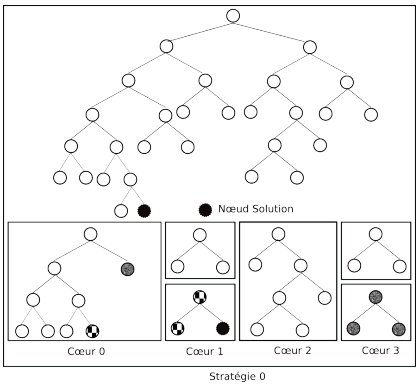


FIGURE 5 – L'équilibrage de charge pour une seule stratégie de recherche en utilisant l'affectation dynamique

La figure 5 présente l'équilibrage de charge obtenu en utilisant l'affectation dynamique. Le lecteur peut facilement voir la différence entre l'équilibrage de charge obtenu par l'affectation dynamique et l'équilibrage de charge obtenu par l'affectation statique présenté dans la figure 4. Il est clair que l'affectation dynamique donne un meilleur équilibrage de charge.

Le problème de l'affectation dynamique est qu'à chaque fois qu'un cœur actif détecte qu'il existe un cœur en attente, il partitionne son sous-arbre avec le cœur en attente sans connaître à priori la taille de son sous-arbre. L'utilisation de ce principe de partitionnement illimité, peut générer beaucoup de sous-arbres, ainsi pour les sous-arbres de petite taille, la résolution séquentielle est parfois plus rapide que la résolution parallèle. Cela vient du fait que le partitionnement des sous-arbres est cher. Pour effectuer un partitionnement, il faut :

- Sauvegarder l'état de toutes les variables dans ce que nous appelons un *Nœud-Bobpp*
- Insérer le *Nœud-Bobpp* dans la FPG
- Pour le cœur en attente qui récupère le *Nœud-Bobpp*, il doit :
  - Ré-initialiser l'état des variables
  - Recommencer la recherche

Pour résoudre le problème de partitionnement illimité, nous proposons d'évaluer la taille des sous-arbres avant d'effectuer le partitionnement. L'évaluation proposée, consiste à calculer le pourcentage des variables non assignées par rapport au nombre total des variables. Si ce pourcentage est plus grand qu'un certain seuil ($\alpha\%$), appelé le seuil de partitionnement, on autorise l'opération de partitionnement. Sinon, une exploration séquentielle est appliquée.

Pour déterminer la valeur du seuil de partitionnement ($\alpha\%$), nous avons effectué quelques expériences, dont les résultats sont présentés dans la figure 6.

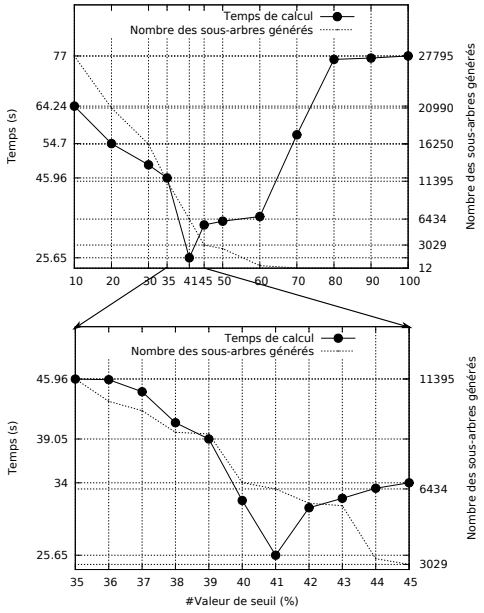


FIGURE 6 – Temps de calcul pour résoudre le problème de Naval Battle (sb_sb_13_13_6_4fzn [11]) qui est un problème de satisfaction de contraintes en utilisant un Portfolio 4 To 12 avec une affectation dynamique

La figure 6 montre le temps de calcul en fonction de la valeur du seuil de partitionnement pour résoudre le problème de Naval Battle (sb_sb_13_13_6_4fzn [11]), qui est un problème de satisfaction de contraintes en utilisant un Portfolio 4 To 12 avec une affectation dynamique. Dans la figure 6 :A, la valeur du seuil varie entre 10% et 100%. On remarque qu'entre 35% et 45% il y a un seuil optimal. Dans la figure 6 :B, nous avons agrandi le résultat en faisant varier le seuil de partitionnement entre 35% et 45%. Effectivement, la valeur optimale du seuil est de 41%. Donc, il existe un seuil optimal pour lequel le temps de calcul est réduit au minimum. Ce seuil représente le meilleur compromis entre un nombre limité des sous-arbres et un temps d'exécution réduit.

Plus le seuil est petit plus on génère un grand nombre de sous-arbres, ce qui facilite l'équilibrage de charge, mais augmente le temps de calcul. Inversement, plus le seuil est grand plus on se rapproche de l'affectation statique où le nombre des sous-arbres est limité et cela donne un mauvais équilibrage de charge.

Le Portfolio N To P avec une affectation dynamique et un seuil de partitionnement fonctionne bien et donne des bons résultats par rapport au Portfolio N To N (modèle classique), présenté dans la sous-section 3.1.

La différence principale entre les stratégies de recherche est la taille de l'arbre généré par chaque stratégie, plus la taille de l'arbre est petite, plus le temps de la résolution est plus petit. Dans le solveur OR-Tools le temps de traitement d'un nœud dans un arbre de recherche généré par n'importe quelle stratégie pour résoudre le même problème de PPC est pratiquement stable (le même temps d'éxecution).

Il est donc intéressant d'adapter le Portfolio N To P avec l'affectation dynamique et un seuil afin de privilégier au moment du partitionnement la stratégie qui est représentée par le plus petit arbre de recherche.

3.3 Portfolio N To P Adaptatif (Adaptation de N Stratégies $\times$ P Cœurs)

Comme toutes les stratégies sont ordonnancées par le même framework parallèle (Bobpp), donc on peut contrôler l'ordonnancement des stratégies entre elles et privilégier la stratégie la plus prometteuse, celle qu'on estime avoir l'arbre de recherche le plus petit, et on lui donne plus de cœurs que les autres stratégies.

Pour déterminer la stratégie qui est candidate à trouver une solution plus rapidement que les autres (la plus prometteuse), à chaque fois qu'un cœur de calcul visite un nœud dans un sous-arbre de n'importe quelle stratégie, il évalue la quantité de travail qui reste à faire. Lorsqu'un cœur actif détecte qu'il y a un cœur en attente, le cœur actif partitionne son sous-arbre si l'évaluation de son sous-arbre est plus grande que le seuil de partitionnement et si sa stratégie de recherche est la meilleure stratégie, c'est à dire que c'est la stratégie pour laquelle il reste le plus petite travail à faire. Sinon, il ne partitionne pas son sous-arbre et il poursuit une exploration séquentielle de l'arbre de recherche. En utilisant ce principe, nous pouvons être sûrs que la stratégie qui partitionne son sous-arbre est la stratégie de recherche la plus prometteuse et on affecte plus de cœurs de calcul à cette stratégie la plus prometteuse.

Pour estimer la quantité de travail qui reste à explorer, nous estimons le nombre maximal des branches qui restent à visiter, et qui est représenté par le produit cartésien de la taille du domaine de valeurs pour chaque variable non assignée. Par exemple, avec juste 100 variables non assignées, et 100 valeurs pour chaque variable, le produit cartésien est 100^{100}, qui est un très grand nombre. Pour éviter les problèmes de capacité et comme cette étude n'est qu'une estimation, nous proposons d'utiliser la somme des valeurs pour chaque variable non assignée, donc pour 100 variables non assignées et avec 100 valeurs pour chaque variable, l'estimation sera 10000.

4 Expérimentation

Pour valider l'approche utilisée dans cette étude, les expériences ont été effectuées en utilisant une machine parallèle Intel Xeon X5650 (2.67 GHz) (12 cœurs physiques) équipée de 48 Go de RAM, avec le système d'exploitation Linux. La version du solveur OR-Tools utilisée comme un moteur de recherche pour résoudre les problèmes de PPC est la version 2727. Tous les problèmes de PPC résolus sont modélisés en utilisant le format FlatZinc et sont issus du MiniZinc Challenge 2012 [11]. FlatZinc est un langage d'entrée de bas niveau utilisé par les solveurs de PPC, il est conçu pour faire une interface entre les problèmes et les solveurs [12]. Le but de ce Minizinc Challenge est de comparer les solveurs de PPC et les différentes méthodes de PPC utilisées pour résoudre le même problème. Dans cette expérience, nous avons résolu 11 problèmes de PPC. 6 sont des problèmes de satisfaction de contraintes et 5 sont des problèmes d'optimisation de contraintes. Les temps de calcul présentés dans cette section sont donnés en secondes et sont une moyenne de minimum de 3 exécutions sans utiliser un temps limite (timeout) pour l'exécution. La valeur du seuil de partitionnement utilisé par le Portfolio N To P est de 40%.

4.1 Pourquoi on utilise le Portfolio ?

Pour résoudre un problème de satisfaction de contraintes, comme le problème des N-Reines [11] de taille 35. L'utilisateur peut utiliser une *stratégie aléatoire* qui choisit aléatoirement une variable de branchement et pour chaque variable, elle assigne une valeur aléatoire. Pour trouver la première solution, la stratégie aléatoire génère un arbre de recherche qui contient 5.505.843 nœuds. Le temps d'exploration de ces nœuds en séquentiel est de 1024,72 secondes. Par contre, si l'utilisateur change la stratégie de recherche et choisit la *stratégie Min Bound*, qui sélectionne la plus petite valeur pour chaque variable non assignée, l'arbre de recherche généré contient un total de 341.593 nœuds et le temps d'exploration de ces nœuds en séquentiel est de 64.14 secondes.

Pour résoudre un problème d'optimisation de contraintes tel que le problème 2D Level Packing [11] (2DLevelPacking_class_1), afin de trouver la solution la plus optimale, l'utilisateur peut choisir la meilleure stratégie trouvée précédemment pour résoudre le problème des N-Reines qui est la stratégie Min Bound, cette stratégie génère un arbre de recherche de 2.745.714 nœuds et le temps de d'exploration de ces nœuds en séquentiel est de 43,90 secondes. D'autre part, si l'utilisateur choisit une autre stratégie de recherche, la *stratégie Impact Base search* [14],

basée sur l'impact des variables et de leurs valeurs, l'arbre de recherche généré contient 31.260 nœuds et le temps d'exploration de ces nœuds en séquentiel est de 1,92 secondes.

Il est donc clair que l'utilisation de la meilleure stratégie a un effet important sur la performance pour résoudre les problèmes de satisfaction et d'optimisation de contraintes. Par exemple, si l'utilisateur utilise la stratégie aléatoire avec 12 cœurs pour résoudre le problème N-Reines et on suppose qu'il a une accélération linéaire, le temps de calcul sera $1024,72/12 = 85{,}39$ secondes. Ce temps de calcul est plus grand que le temps de calcul obtenu en séquentiel en utilisant la stratégie Min Bound (64.14 secondes).

En utilisant un solveur Portfolio, nous pouvons être sûr que la performance obtenue est la même que la performance de la meilleure stratégie.

4.2 Les Performances Obtenues en Utilisant la Parallélisation Portfolio

Dans ce qui suit, l'accélération est calculé par rapport à un Portfolio *N* To *N*.

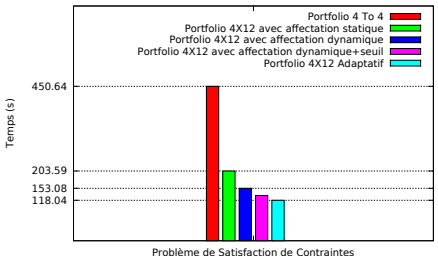


FIGURE 7 – Comparaison entre 5 modèles de Portfolio pour résoudre le problème de Naval Battle [11] (sb_sb_13_13_5_1fzn) qui est un problème de satisfaction de contraintes

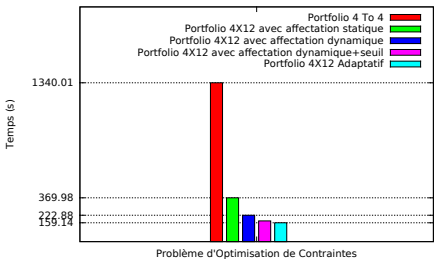


FIGURE 8 – Comparaison entre 5 modèles de Portfolio pour résoudre le problème de Pattern Set Mining [11] (pattern_set_mining_k1_germancredit) qui est un problème d'optimisation de contraintes

Les figures 7 et 8 montrent une comparaison entre les 5 modèles de Portfolio : *N* To *N* avec 4 stratégies et 4 cœurs, *N* To *P* avec 4 stratégies et 12 cœurs avec affectation statique, dynamique et dynamique avec un seuil de partitionnement. Le dernier modèle est le Portfolio *N* To *P* Adaptatif avec 4 stratégies et 12 cœurs. Cette comparaison est effectuée en résolvant le problème de Naval Battle [11] (sb_sb_13_13_5_1), qui est un problème de satisfaction de contraintes et le problème de Pattern Set Mining [11] (pattern_set_mining_k1_germancredit), qui est un problème d'optimisation de contraintes.

La performance des Portfolios *N* To *N* et *N* To *P* en utilisant une affectation statique est limitée, car ils ont une faible accélération par rapport aux différentes versions de Portfolio *N* To *P* avec affectation dynamique et le Portfolio Adaptatif. Le meilleur Portfolio est le Portfolio Adaptatif.

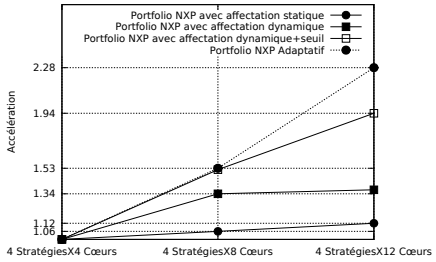


FIGURE 9 – Comparaison d'accélération moyenne pour résoudre 6 problèmes de satisfaction de contraintes en utilisant 4 modèles de Portfolio (4 stratégies de recherche pour chaque modèle)

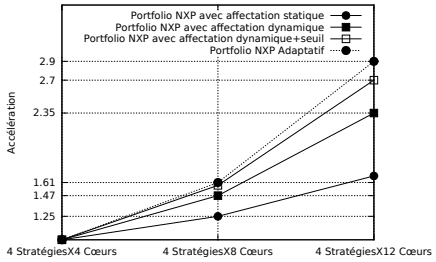


FIGURE 10 – Comparaison d'accélération moyenne pour résoudre 5 problèmes d'optimisation de contraintes en utilisant 4 modèles de Portfolio (4 stratégies de recherche pour chaque modèle)

Les figures 9 et 10 montrent une accélération moyenne pour résoudre 6 problèmes de satisfaction de contraintes et 5 problèmes d'optimisation de contraintes en utilisant 4 modèles de Portfolio *N* To *P*, chaque modèle de Portfolio utilise 4 stratégies de

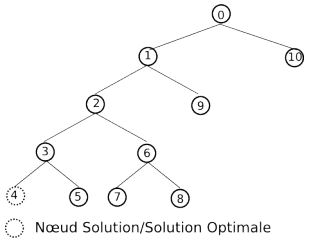


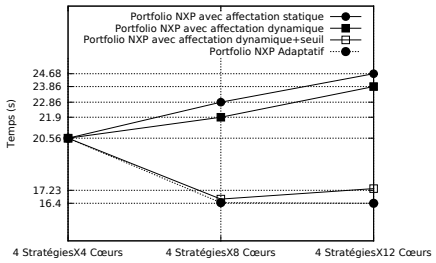
FIGURE 11 – Exemple d'arbre de recherche qui donne une mauvaise accélération

recherche et le nombre de cœurs varie entre 4, 8 et 12 cœurs.

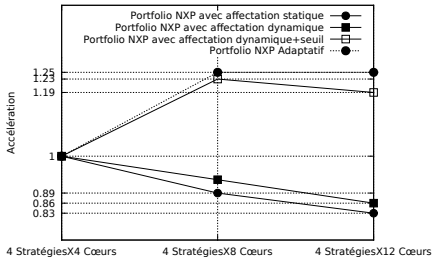
Parfois, le nœud solution est le nœud le plus à gauche dans l'arbre de recherche, tel que présenté dans la figure 11. Dans ce cas, le Portfolio N To P ne donne pas une bonne accélération, comme le montrent les figures 12 et 14. La figure 12 :a *(resp. 12 :b)* montre le temps de calcul *(resp. accélération)* pour résoudre le problème de Naval Battle(sb_sb_15_15_7_4) [11], qui est un problème de satisfaction de contraintes et qui utilise 4 stratégies de recherche pour chaque modèle de Portfolio. Le tableau de la figure 12 présente le nombre des nœuds OR-Tools visités par chaque modèle de Portfolio. La figure 13 montre une parallélisation de la meilleure stratégie pour résoudre le problème de Naval Battle (sb_sb_15_15_7_4). La mauvaise accélération vient de la parallélisation de l'arbre de recherche (anomalie de recherche).

La figure 14 :a *(resp. 14 :b)* montre le temps de calcul *(resp. accélération)* pour résoudre le problème de Pattern Set Mining (pattern_set_mining_k1_segment) [11], qui est un problème d'optimisation de contraintes en utilisant 4 stratégies de recherche pour chaque Portfolio. Le tableau de la figure 14 représente le nombre de nœuds OR-Tools visités par chaque modèle de Portfolio. La figure 15 montre une parallélisation de la meilleure stratégie pour résoudre le problème de Pattern Set Mining (pattern_set_mining_k1_segment). Le résultat est similaire au problème de satisfaction de contraintes, et la mauvaise accélération provient de la parallélisation de l'arbre de recherche (anomalie de recherche).

Il est possible de trouver un nœud solution au premier niveau de l'arbre de recherche, tel que présenté dans la figure 16. Dans ce cas, le Portfolio N To P donne une bonne accélération, comme le montrent les figures 17 et 18. La figure 17 :a *(resp. 17 :b)* montre le temps de calcul *(resp. accélération)* pour résoudre le problème de Naval Battle(sb_sb_13_13_5_1) [11], qui est un problème de satisfaction de contraintes. Le tableau de la figure 17 représente le nombre de nœuds OR-Tools visités par chaque modèle de Portfolio.



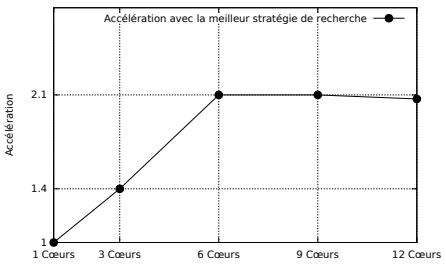
(a)



(b)

# Cœurs	Modèles de Portfolio				
	N To N	N To P			
	4 To 4	Statique	Dynamique	Dynamique avec seuil	Adaptatif
4	488,316	-	-	-	-
8	488,316	578,022	575,354	565,322	508,711
12	488,316	715,017	669,515	607,417	519,192

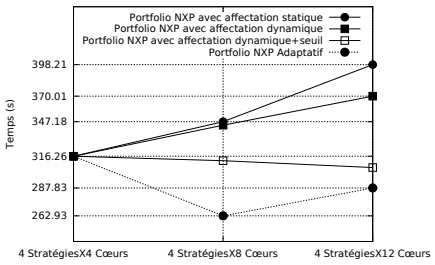
FIGURE 12 – Pire des cas (Worst case) pour résoudre le problème de Naval Battle(sb_sb_15_15_7_4) [11] qui est un problème de satisfaction de contraintes en utilisant 4 stratégies de recherche pour chaque Portfolio



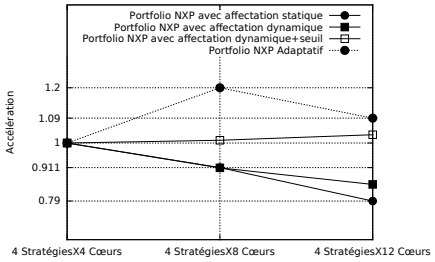
(a)

Number of nodes	Nombre des cœurs				
	1	3	6	9	12
	127,239	186,401	213,420	338,783	293,066

FIGURE 13 – Parallélisation de l'arbre de recherche généré par la meilleure stratégie pour résoudre le problème de Naval Battle (sb_sb_15_15_7_4) [11] qui est problème de satisfaction de contraintes



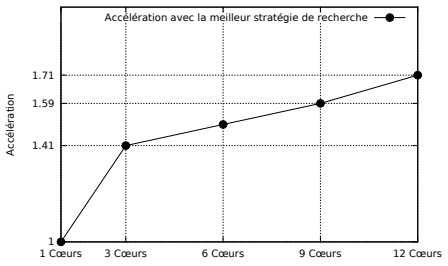
(a)



(b)

# Cœurs	Modèles de Portfolio				
	N To N	N To P			
	4 To 4	Statique	Dynamique	Dynamique avec seuil	Adaptatif
4	325,948	-	-	-	-
8	325,948	325,657	353,283	344,933	310,596
12	325,948	461,803	421,402	408,897	334,440

FIGURE 14 – Pire des cas (Worst case) pour résoudre le problème de Pattern Set Mining (pattern_set_mining_k1_segment) [11] qui est un problème d'optimisation de contraintes en utilisant 4 stratégies de recherche pour chaque Portfolio



(a)

Number of nodes	Nombre des cœurs				
	1	3	6	9	12
	82,281	109,000	135,548	137,828	141,819

FIGURE 15 – Parallélisation de l'arbre de recherche généré par la meilleure stratégie pour résoudre le problème de Pattern Set Mining (pattern_set_mining_k1_segment) [11] qui est un problème d'optimisation de contraintes

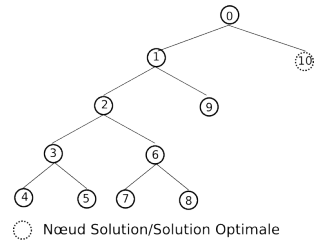
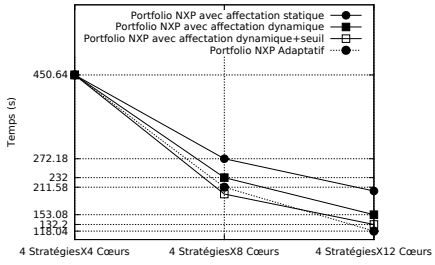
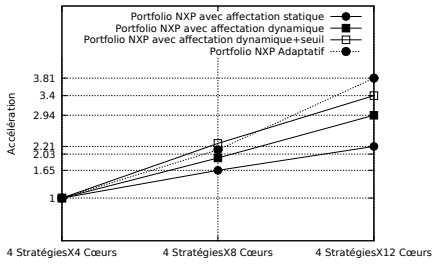


FIGURE 16 – Exemple d'arbre de recherche qui donne une bonne accélération



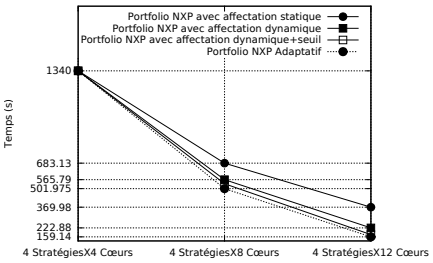
(a)



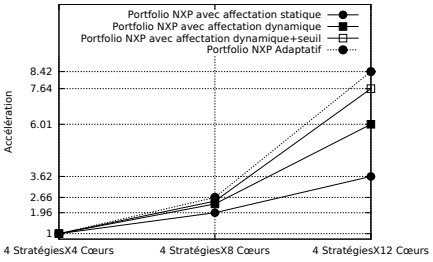
(b)

# Cœurs	Modèles de Portfolio				
	N To N	N To P			
	4 To 4	Statique	Dynamique	Dynamique avec seuil	Adaptatif
4	11,097,805	-	-	-	-
8	11,097,805	7,119,938	6,666,043	6,582,847	6,009,770
12	11,097,805	7,017,854	6,287,693	607,158	5,168,337

FIGURE 17 – Meilleur cas (Best case) pour résoudre le problème de Naval Battle(sb_sb_13_13_5_1) [11] qui est un problème de satisfaction de contraintes en utilisant 4 stratégies de recherche pour chaque Portfolio



(a)



(b)

# Cœurs	Modèles de Portfolio				
	N To N	N To P			
	4 To 4	Statique	Dynamique	Dynamique avec seuil	Adaptatif
4	3,925,159	-	-	-	-
8	3,925,159	1,633,120	1,596,446	1,535,643	1,517,335
12	3,925,159	333,404	306,790	267,996	232,213

FIGURE 18 – Meilleur cas (Best case) pour résoudre le problème de Patter Set Mining (pattern_set_mining_k1_germancredit) [11] qui est un problème d'optimisation de contraintes en utilisant 4 stratégies de recherche pour chaque Portfolio

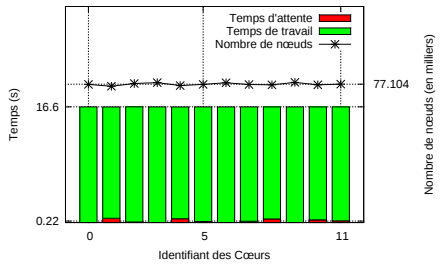


FIGURE 19 – Équilibrage de charge pour résoudre le problème de Naval Battle (sb_sb_15_15_7_0) [11] qui est un problème de satisfaction de contraintes avec un Portfolio Adaptatif de 4 stratégies et 12 cœurs

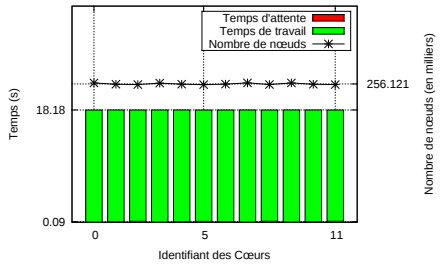


FIGURE 20 – Équilibrage de charge pour résoudre le problème de Open Stacks (open_stacks_01_problem_15_15) [11] qui est un problème d'optimisation de contraintes avec un Portfolio Adaptatif de 4 stratégies et 12 cœurs

La figure 18 :a *(resp. 18 :b)* montre le temps de calcul *(resp. accélération)* pour résoudre le problème de Pattern Set Mining (pattern_set_mining_k1_germancredit) [11], qui est un problème d'optimisation de contraintes. Le tableau de la figure 18 représente le nombre de nœuds OR-Tools visités par chaque modèle de Portfolio. Pour les deux problèmes, chaque modèle de Portfolio utilise 4 stratégies de recherche. Les figures 19 et 20 montrent le comportement des cœurs de calcul lorsque nous utilisons le Portfolio Adaptatif. Le résultat est que les temps de calcul et d'attente sont répartis de manière équitables entre les cœurs de calcul. De plus, tous les cœurs ont visité le même nombre de nœuds OR-Tools.

5 Conclusion et Perspectives

Cet article présente un Portfolio parallèle N To P pour résoudre les problèmes de satisfaction et d'optimisation de contraintes en utilisant le solveur OR-Tools au-dessus du framework parallèle Bobpp. Le meilleur Portfolio N To P est le Portfolio Adaptatif qui privilège la stratégie la plus prometteuse.

Une première perspective est d'effectuer plusieurs expériences pour déterminer automatiquement la valeur du seuil de partitionnement utilisé par l'affectation dynamique en fonction des problèmes de PPC.

L'utilisation du framework parallèle Bobpp donne une bonne accélération sur des architectures à mémoire partagée. Ces résultats sont obtenus avec plusieurs types de problèmes d'optimisation combinatoire sur différents ordinateurs. Bobpp a également une parallélisation sur des architectures à mémoire distribuée version mixte MPI/Pthreads. Une deuxième perspective est de proposer une parallélisation distribuée pour le Portfolio Adaptatif.

Enfin, il serait intéressant de retourner à l'utilisateur toujours la même solution quel que soit le mode d'exécution. En séquentiel, c'est automatique, car c'est la première solution trouvée. Cependant, en parallèle la première solution trouvée n'est pas nécessairement la même solution que la solution retournée en séquentiel. Une dernière perspective serait de proposer un Portfolio Adaptatif déterministe qui retourne à l'utilisateur toujours la même solution si cette fonctionnalité est

demandée par l'utilisateur.

Références

[1] Arbelaez Alejandro, Hamadi Youssef, and Sebag Michèle. Online Heuristic Selection in Constraint Programming, 2009. International Symposium on Combinatorial Search - 2009.

[2] Bertrand Le Cun, Tarek Menouer, and Pascal Vander-Swalmen. Bobpp. `http://forge.prism.uvsq.fr/projects/bobpp`.

[3] e. o'mahony, Emmanuel Hebrard, Alan Holland, and Conor Nugent. Using case-based reasoning in an algorithm portfolio for constraint solving. In *IRISH CONFERENCE ON ARTIFICIAL INTELLIGENCE AND COGNITIVE SCIENCE*, 2008.

[4] Xie Feng and Davenport Andrew. Solving scheduling problems using parallel message-passing based constraint programming. In *Proceedings of the Workshop on Constraint Satisfaction Techniques for Planning and Scheduling Problems COPLAS*, pages 53–58, 2009.

[5] Francois Galea and Bertrand Le Cun. Bob++ : a framework for exact combinatorial optimization methods on parallel machines. In *International Conference High Performance Computing & Simulation 2007 (HPCS'07) and in conjunction with The 21st European Conference on Modeling and Simulation (ECMS 2007)*, pages 779–785, June 2007.

[6] Diarmuid Grimes and Richard J. Wallace. Sampling strategies and variable selection in weighted degree heuristics. In *13th International Conference on Principles and Practice of Constraint Programming 2007, Providence, RI, USA*, 2007.

[7] Joxan Jaffar, Andrew E. Santosa, Roland H. C. Yap, and Kenny Qili Zhu. Scalable distributed depth-first search with greedy work stealing. In *ICTAI*, pages 98–103, 2004.

[8] Stephan Kottler and Michael Kaufmann. SArTagnan - A parallel portfolio SAT solver with lockless physical clause sharing. In *Pragmatics of SAT*, 2011.

[9] Tarek Menouer and Bertrand Le Cun. Anticipated dynamic load balancing strategy to parallelize constraint programming search. In *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2013 IEEE 27th International*, pages 1771–1777, May 2013.

[10] Laurent Michel, Andrew See, and Pascal Van Hentenryck. Transparent parallelization of constraint programming. *INFORMS JOURNAL ON COMPUTING*, 21(3) :363–382, 2009.

[11] Minizinc challenge `http://www.minizinc.org/challenge2012/`, 2012.

[12] NICTA. Specification of zinc and minizinc. Technical report, Victoria Research Lab, Melbourne, Australia, 2011.

[13] Laurent Perron. Search procedures and parallelism in constraint programming. *International Conference on Principles and Practice of Constraint Programming*, 1999.

[14] Philippe Refalo. Impact-based search strategies for constraint programming. In *CP*, pages 557–571, 2004.

[15] Jean-Charles Regin, Mohamed Rezgui, and Arnaud Malapert. Embarrassingly parallel search. In *19th International Conference on Principles and Practice of Constraint Programming 2013 Uppsala Sweden*, 2013.

[16] Olivier Roussel. ppfolio. `http://www.cril.univ-artois.fr/~roussel/ppfolio/`.

[17] Vincent Vidal, Lucas Bordeaux, and Youssef Hamadi. Adaptive k-parallel best-first search : A simple but efficient algorithm for multi-core domain-independent planning, 2010.

[18] Lin Xu, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. Satzilla : Portfolio-based algorithm selection for sat. *J. Artif. Int. Res.*, 32(1) :565–606, June 2008.

Une Approche par Décomposition pour la Découverte de Motifs Discriminants sur Données Séquentielles

Vincent Vigneron[1] **David Lesaint**[1] **Deepak Mehta**[2] **Barry O'Sullivan**[2]

[1] LERIA, Université d'Angers, France

[2] INSIGHT Centre for Data Analytics Department of Computer Science, University College Cork, Ireland

{david.lesaint,vincent.vigneron}@info.univ-angers.fr {d.metha,b.osullivan}@4c.ucc.ie

Résumé

La Programmation par Contraintes est une approche récente en Fouille de Données qui cherche à pallier au manque de flexibilité des algorithmes dédiés. Cet article s'inscrit dans cette démarche et traite de la recherche de motifs discriminants dans une base de séquences étiquetées positives ou négatives. Nous définissons un motif comme un ensemble de c-blocs où un c-bloc correspond à l'enracinement d'une sous-séquence commune aux séquences positives et dont le nombre de caractères libres entre deux sous-chaînes reste constant d'une séquence à l'autre. Nous reformulons des contraintes classiques de fouille pour ce modèle de motif et présentons de nouvelles contraintes qui éliminent des motifs redondants ou répondent à des besoins applicatifs. Nous modélisons le calcul de motif discriminant comme un problème d'optimisation paramétré par deux ensembles de contraintes dites positives et négatives. Un motif solution doit satisfaire toutes ces contraintes sur la classe positive et exclure le plus grand nombre de séquences négatives, c'est-à-dire, toute séquence où l'on ne peut l'étendre sans violer de contraintes négatives. Nous nous focalisons sur la classe de problèmes visant à calculer des motifs fermés, sans réplications et couvrant minimalement la classe négative. Nous présentons une méthode qui construit des c-blocs fermés de manière ad-hoc puis détermine un motif solution par résolution d'un problème de couverture minimum par ensembles. Des résultats expérimentaux obtenus sur des bases de séquences protéiques en démontrent l'efficacité.

1 Introduction

La fouille de données revêt une importance majeure dans de nombreux domaines allant de la Bio-informatique à la Cyber-sécurité. Elle vise à extraire des caractéristiques propres à un ensemble de données, par exemple, la présence d'itemsets fréquents dans une base de transactions. De nombreux algorithmes de fouille ont été conçus selon le type de données à traiter (itemsets, séquences, données étiquetées ou non, etc.) ou le calcul à mener (apprentissage de concept, partitionnement, etc). Les algorithmes dédiés étant difficilement adaptables lorsque les besoins évoluent, des approches fondées sur la Programmation par Contraintes (PPC) ont été proposées pour une formulation plus déclarative des problèmes de fouille [3, 4, 9], l'objectif étant de concilier flexibilité de modélisation et efficacité de calcul.

Cet article s'inscrit dans cette démarche et aborde le problème de la recherche de motifs discriminants dans une base de séquences prépartitionnée en classes. Ce problème est d'intérêt en Bio-informatique pour valider a posteriori le partitionnement expert d'un jeu de séquences protéiques en signant chaque classe par un motif exclusif. Sans perte de généralité, nous considérons une base bi-partitionnée en *séquences positives* et *séquences négatives*. L'objectif est alors d'identifier un motif commun et exclusif à la classe des séquences positives. Nous définissons un motif comme un ensemble de *c-blocs* (blocs de classe). Chaque c-bloc correspond à l'enracinement d'une sous-séquence commune aux séquences positives et dont le nombre de caractères libres entre deux sous-chaînes est invariant d'une séquence à l'autre.

Ce modèle de motif permet de distinguer contraintes sur

c-blocs (par exemple, expressions régulières) et contraintes entre c-blocs (par exemple, absence de répétitions) sans préjuger d'un traitement unifié ou séparé. Nous reformulons d'une part les contraintes classiques de fermeture et couverture. Ces contraintes ont été formalisées dans des approches PPC pour d'autres types de calcul : motifs unaires [3] et n-aires [4] à base d'itemsets, motifs unaires pour séquence simple [1, 2] ou séquences multiples [9]. Nous présentons d'autre part des contraintes de non-inclusion et de non-réplication entre c-blocs. Ces dernières visent à éliminer certains motifs redondants à l'instar des contraintes de fermeture et de couverture.

Nous modélisons la recherche de motif discriminant sous la forme d'un problème d'optimisation sous contraintes (Motif Discovery Problem) paramétré par deux ensembles de contraintes dites positives et négatives. Un motif solution doit satisfaire toutes ces contraintes sur la classe positive et exclure le plus grand nombre de séquences négatives. L'exclusivité se mesure au nombre de séquences négatives sur lesquelles on ne peut étendre le motif sans violer l'une des contraintes négatives. Un MDP tolère des critères additionnels basés sur des mesures telle que la *variabilité* de motif (nombre maximum de caractères libres successifs au sein des c-blocs).

Le cadre MDP laisse toute flexibilité dans le choix et la répartition des contraintes à traiter. Les propriétés de contraintes (anti-monotonie, subsomption, etc) peuvent être exploitées à cet égard. Par exemple, toute extension sur la classe négative d'un motif sans réplications étant nécéssairement sans réplications, il est inutile de considérer la contrainte de non-réplication pour la preuve d'exclusivité. De même, tout motif sans réplications étant nécéssairement sans inclusions, cette dernière contrainte peut être ignorée si la non-réplication est requise.

Nous nous focalisons sur la classe de problèmes MDP qui vise au calcul de motifs fermés sur la classe positive, sans réplications et couvrant minimalement la classe négative (Replication-free MDP). Nous montrons que, sous certaines conditions portant sur la fonction objectif, tout RMDP satisfaisable admet un motif solution constitué de c-blocs maximaux pour la relation de réplication. Pour le cas général, nous présentons une méthode de résolution en deux étapes qui consiste à générer un ensemble de c-blocs fermés et sans inclusions puis à déterminer un motif solution. Nous considérons précisément le cas où la fonction objectif combine les critères suivants par ordre de priorité : exclusion maximum, variabilité minimum et cardinalité minimum. La méthode utilise un algorithme ad-hoc pour le calcul de c-blocs et ramène la seconde étape à la résolution d'un problème de couverture minimum par ensembles.

Des résultats expérimentaux obtenus sur deux bases de séquences protéiques [6, 7, 8] attestent de son efficacité. Ces résultats permettent par ailleurs d'exhiber des motifs non séquentialisables, autrement dit, des motifs incalculables à l'aide d'algorithmes d'alignement de séquences. En ce sens, ils légitiment le cadre MDP qui laisse à l'utilisateur le libre choix des contraintes à satisfaire selon les résultats obtenus à chaque étape d'un processus de fouille.

Le reste de l'article s'organise comme suit. La section 2 présente le modèle de motif à base de c-blocs et formalise les problèmes MDP et RMDP. La section 3 décrit la méthode de résolution de RMDP. La section 4 présente les résultats expérimentaux et en fournit une analyse. La section 5 conclut.

2 Définitions

Soit Σ un alphabet composé d'un nombre fini de caractères dits solides, une séquence sur Σ est définie par une suite de caractères $s_1, \ldots, s_n \in \Sigma^n$ ($n \in \mathbb{N}^*$). La longueur d'une séquence s est notée $|s|$. Une base de séquences bipartitionnée D est l'union de deux ensembles disjoints D^+ et D^- de séquences sur Σ ($D \subseteq \Sigma^*$ et $D = D^+ \cup D^-$). $D+$ (respectivement, D^-) est la classe des séquences dites positives (resp., négatives). Un hash est un caractère additionnel $\#$ qui n'appartient pas à Σ ($\# \notin \Sigma$). Un patron est une séquence sur $\Sigma \cup \{\#\}$ qui démarre et se termine par un caractère solide et est de longueur supérieure ou égale à 2. Les définitions qui suivent font référence à une base de séquences D sur un alphabet Σ. On notera $[n]$ l'intervalle $\{i \mid 1 \leqslant i \leqslant n\}$ pour $n \in \mathbb{N}$ et $[t]$ l'ensemble $[|t|]$ pour toute séquence, patron ou ensemble t.

Définition 1 (Patron) *Un patron est une séquence $p \in \Sigma.(\Sigma \cup \{\#\})^*.\Sigma$. On note p_i le $i^{ème}$ caractère de p pour tout $i \in [p]$, $\pi(p) = \{k \mid p_k \neq \#\}$ l'ensemble des positions des caractères solides dans p et $\pi(p)(i)$ la position du $i^{ème}$ caractère solide de p dans p pour tout $i \in [\pi(p)]$.*

Un patron peut apparaître plus d'une fois dans une même séquence. Par exemple, le patron `A##C#C` est enraciné aux positions 1 et 3 dans la séquence `ADACDCEC`. Un patron peut également apparaître dans différentes séquences de la base de données et l'on appelle couverture d'un patron l'ensemble des séquences dans lesquelles il est enraciné. Un patron est dit positif si sa couverture inclut la classe positive.

Définition 2 (Enracinement et couverture d'un patron) *Un patron p est enraciné dans une séquence s à la position $l \in [s]$, noté $p \subseteq_l s$, si $p_k = s_{l+k-1}$ ou $p_k = \#$ pour tout $k \in [p]$. La couverture d'un patron p, notée $\varphi(p)$, est l'ensemble $\varphi(p) = \{s \in D \mid \exists l \in [s],\ p \subseteq_l s\}$.*

Un bloc correspond à l'enracinement d'un patron dans une séquence. Un c-bloc correspond à l'enracinement d'un patron positif sur toute sa couverture. Un c-bloc résulte

donc du choix d'un bloc par séquence figurant dans la couverture du patron. Un motif est un ensemble de c-blocs et l'on définit sa couverture comme étant l'intersection des couvertures de ses c-blocs. Par définition, tout motif couvre la classe positive.

Définition 3 (Bloc, c-bloc et motif) *Un bloc est un triplet (p, l, s) où s est une séquence de D et p un patron enraciné à la position l dans s ($p \subseteq_l s$). Un c-bloc est une paire (p, δ) où p est un patron positif et δ est une fonction qui enracine p sur sa couverture : $D^+ \subseteq \varphi(p)$, $\delta \in (\varphi(p) \to \mathbb{N}^*)$ et $(p, \delta(s), s)$ est un bloc pour tout $s \in \varphi(p)$. La couverture d'un c-bloc $c = (p, \delta)$ est la couverture de son patron : $\varphi(c) = \varphi((p, \delta))$. Un motif est un ensemble de c-blocs. La couverture d'un motif Π est l'ensemble $\varphi(\Pi) = \bigcap_{c \in \Pi} \varphi(c)$.*

La Figure 1 illustre patrons et blocs pour une base de trois séquences. Par exemple, AA est un patron positif ; $(\texttt{AA}, 6, s_1)$, $(\texttt{AA}, 8, s_2)$ et $(\texttt{AA}, 1, s_3)$ sont des blocs pour ce patron et $(\texttt{AA}, \{6, 8, 1\})$ est donc un c-bloc possible [1]. $(\texttt{AAC\#E}, \{1, 1\})$ est un autre exemple de c-bloc. A l'inverse, $(\texttt{YE}, \{4\})$ n'est pas un c-bloc car le patron YE n'est pas positif. $\{(\texttt{A\#C}, \{1, 1, 1\}), (\texttt{AAC}, \{1, 1, 1\}), (\texttt{AAC\#E}, \{1, 1\}), (\texttt{AA}, \{6, 8, 1\})\}$ est un exemple de motif dont la couverture est D^+.

Nous recherchons des motifs qui sont communs et exclusifs à la classe positive, c'est-à-dire qui satisfont les contraintes prescrites sur la classe positive, et qui violent au moins une contrainte lorsqu'ils sont étendues aux séquences négatives. Chaque contrainte sur motif s'interprète donc relativement au choix d'un ensemble de séquences selon le calcul : classe positive pour le calcul de motif commun et sur-ensemble de la classe positive pour la preuve d'exclusivité.

Définition 4 (Contrainte sur motifs) *Soit C l'ensemble des c-blocs pour une base D. Une contrainte r sur motifs de D est une fonction $r : 2^D \times 2^C \to \{false, true\}$. On dit qu'un motif $\Pi \subseteq C$ satisfait une contrainte r sur $X \subseteq D$ si $r(X, \Pi) = true$.*

	s_1	s_2	s_3
Séquences	AACYEAA	AACZEZZAA	AAC
Blocs	A#C	A#C	A#C
	AAC	AAC	AAC
	AAC#E	AAC#E	
	AA	AA	AA

FIGURE 1 – Exemple de blocs pour une base de séquences $D^+ = \{s_1, s_2\}$ et $D^- = \{s_3\}$.

1. Nous supposons un ordre sur les séquences de D et nous représentons la fonction d'enracinement d'un c-bloc par son image ordonnée par rapport à D.

Nous présentons ci-dessous les contraintes de non-inclusion, non-réplication, couverture et fermeture. Les contraintes de non-inclusion et de non-réplication portent sur des couples de c-blocs alors que les contraintes de couverture et de fermeture portent sur un seul c-bloc à la fois. Ces contraintes permettent d'éliminer des motifs redondants en se basant sur les patrons ou les blocs de c-blocs.

La réplication est une contrainte de subsomption sur les patrons. Précisément, un patron p réplique un patron p' si p est obtenu à partir de p' en y remplaçant des hashs par des caractères solides ou en l'étendant à gauche ou à droite avec des caractères solides ou des hashs. Par exemple, le patron AAC#E réplique AAC qui lui même réplique A#C. L'inclusion est une contrainte de subsomption sur les blocs. Précisément, un bloc b inclut un bloc b' si b est obtenu à partir de b' en y remplaçant des hashs par des caractères solides ou en ajoutant des caractères solides ou des hashs à gauche ou à droite. Par exemple, le bloc $(\texttt{AAC\#E}, 1, s_1)$ inclut $(\texttt{AAC}, 1, s_1)$ qui lui même inclut $(\texttt{A\#C}, 1, s_1)$ en Figure 1.

Définition 5 (Inclusion et réplication de blocs) *Soit $b = (p, l, s)$ et $b' = (p', l', s)$ deux blocs pour une séquence $s \in D$. b inclut b', noté $b \sqsupseteq b'$, si $\{k + l \mid k \in \pi(p)\} \supseteq \{k + l' \mid k \in \pi(p')\}$. b réplique b', noté $b \succeq b'$, si $\exists b'' = (p, l'', s),\ b'' \sqsupseteq b'$.*

Il y a réplication entre deux c-blocs s'il y a réplication entre leurs patrons. Il y a inclusion entre deux c-blocs sur un ensemble de séquences $X \subseteq D$ si ces c-blocs ont une séquence commune dans X sur laquelle l'inclusion se vérifie entre leurs blocs.

Définition 6 (Inclusion et réplication de c-blocs) *Soit $c = (p, \delta)$ et $c' = (p', \delta')$ deux c-blocs. c réplique c', noté $c \succeq c'$, si $\exists s \in D^+,\ (p, \delta(s), s) \succeq (p', \delta'(s), s)$. c inclut c' sur $X \subseteq D$, noté $c \sqsupseteq_X c'$, si $\exists s \in X \cap \varphi(c) \cap \varphi(c'),\ (p, \delta(s), s) \sqsupseteq (p', \delta'(s), s)$.*

La contrainte de couverture d'un ensemble $X \subseteq D$ par un motif stipule que la couverture du motif inclue X. La contrainte de fermeture stipule que les c-blocs du motif ne puissent pas être étendus, c'est à dire, ne puissent pas être répliqués tout en préservant leurs enracinements. Précisément, un c-bloc est fermé sur $X \subseteq D$ s'il n'existe aucun c-bloc l'incluant strictement sur toute séquence commune dans X. Par exemple, le c-bloc $(\texttt{AAC}, \{1, 1, 1\})$ en Figure 1 n'est pas fermé sur D^+ car $(\texttt{AAC\#E}, \{1, 1, 1\})$ l'inclut strictement sur toute séquence positive mais il est fermé sur D.

Définition 7 (Fermeture de c-blocs) *Un c-bloc (p, δ) est fermé sur $X \subseteq D$ s'il n'existe pas de c-bloc (p', δ') tel que $(p', \delta'(s), s) \sqsupsetneq (p, \delta(s), s)$ pour tout $s \in X \cap \varphi((p, \delta)) \cap \varphi((p', \delta'))$.*

Nous formalisons maintenant ces contraintes sur motifs relativement à un ensemble de séquences $X \subseteq D$. Un motif couvre X si sa couverture inclut X. Un motif est fermé sur X si ses c-blocs sont fermés sur X. Un motif est sans-réplications sur X si il n'y a pas de réplication entre ses c-blocs. Un motif est sans-inclusions sur X s'il n'y pas d'inclusion entre ses c-blocs sur X. Formellement [2],

$$couverture_X(\Pi) \Leftrightarrow X \subseteq \varphi(\Pi)$$
$$fermeture_X(\Pi) \Leftrightarrow \forall c \in \Pi,\ c \textit{ est fermé sur } X$$
$$non-replication_X(\Pi) \Leftrightarrow \forall c, c' \in \Pi,\ c \geq c' \Rightarrow c = c'$$
$$non-inclusion_X(\Pi) \Leftrightarrow \forall c, c' \in \Pi,\ c \sqsupseteq_X c' \Rightarrow c = c'$$

Le motif $\{(\texttt{AAC}, \{1,1,1\}), (\texttt{AAC\#E}, \{1,1\})\}$ en Figure 1 couvre D^+ mais ne couvre pas D et il n'est ni fermé, ni sans-inclusions sur sa couverture. Le motif $\{(\texttt{AAC}, \{1,1,1\}), (\texttt{AA}, \{6,8,1\})\}$ couvre D et est sans-inclusions mais il n'est ni fermé, ni sans-réplications sur D^+. Le motif $\{(\texttt{AAC\#E}, \{1,1\}), (\texttt{AA}, \{6,8,1\})\}$ couvre D^+ et il est fermé et sans-inclusions sur sa couverture mais il n'est pas sans-réplications.

D'autres contraintes peuvent être envisagées : contraintes sur patrons (par exemple, restreindre les caractères solides admissibles, borner le nombre de hashs, et plus généralement imposer des contraintes sous la forme d'expressions régulières [9]), contraintes sur c-blocs (par exemple, classes de substitutions autorisées pour tout hash), ou contraintes entre c-blocs (par exemple, contraintes sur le nombre de réplications ou de répétitions de c-blocs).

Le fait d'opérer sur des données séquentielles permet aussi de contraindre le positionnement ou l'ordonnancement interne de motif. En effet, chaque bloc correspond à un intervalle de positions et des relations tirées d'algèbres de points ou d'intervalles peuvent être utilisées pour imposer un ordonnancement cohérent des c-blocs d'un motif sur sa couverture. On peut notamment rechercher des motifs "séquentiels" en imposant un ordre total entre c-blocs par le biais de contraintes de précédence.

Ce calcul est une approche possible (non développée ici) au problème d'alignement de séquences multiples en Bio-informatique [5]. Un alignement s'obtient par une série de substitutions, suppressions ou insertions de caractères sur chaque séquence. Or tout c-bloc détermine un alignement possible des séquences positives, chaque caractère libre pouvant s'interpréter comme un point de substitution (par exemple, un point de mutation évolutionnaire dans une séquence protéique). D'autre part, toute variation d'écart entre c-blocs successifs d'un motif sur différentes séquences peut s'interpréter comme une série de suppressions ou d'insertions (par exemple, suppressions d'acides aminés dans une séquence protéique).

2. Pour une contrainte sur motif r, nous notons $r_X(\Pi)$ pour signifier $r(X, \Pi) = true$ et $\neg r_X(\Pi)$ pour signifier $r(X, \Pi) = false$.

Nous présentons maintenant le cadre MDP pour le problème de recherche de motifs discriminants. Un MDP est paramétré par un ensemble de contraintes et requiert qu'un motif solution satisfasse toutes ces contraintes sur la classe positive et qu'il exclue le plus grand nombre de séquences négatives. Un motif exclut une séquence négative si on ne peut "l'étendre" à cette séquence sans violer l'une des contraintes. Autrement dit, tout motif identique sur la classe positive viole nécessairement une contrainte lorsque l'interprétation incorpore la séquence négative.

Définition 8 (Mesure d'exclusion) *Soient Π et Π' deux motifs, on note $\Pi =_+ \Pi'$ s'il existe une bijection $f : \Pi \rightarrow \Pi'$ vérifiant $\forall (p, \delta) \in \Pi,\ f((p, \delta)) = (p', \delta') \Rightarrow (p = p' \wedge \delta_{|D^+} = \delta'_{|D^+})$. Soit R un ensemble de contraintes sur motifs, $exc_R(\Pi) = |\{s \in D^- \mid \forall \Pi' =_+ \Pi, \exists r \in R, \neg r(\varphi(\Pi) \cup \{s\}, \Pi')\}|$ est la mesure d'exclusion de Π et $inc_R(\Pi) = |D^-| - exc_R(\Pi)$ est sa mesure d'inclusion.*

Pour plus de flexibilité, un MDP permet de limiter l'ensemble des contraintes à considérer pour la measure d'exclusion. En effet, les contraintes monotones sur ensembles de séquences ou dont la sémantique ne dépend pas de la couverture de motif peuvent être écartées sans risques. C'est le cas la contrainte de non-réplication qui est indépendante de la couverture de motif et de la contrainte de fermeture qui est monotone (un motif fermé sur X l'est pour tout $Y \supseteq X$). A l'inverse, les contraintes de couverture et de non-inclusion sont anti-monotones et cette dernière sous-somme la la contrainte de non-réplication.

Définition 9 (MDP) *Un problème de découverte de motifs est un n-uplet $P = (D, R^+, R^-, <)$ où $D = D^+ \uplus D^-$ est une base de séquences bi-partitionnée, R^+ et R^- sont des ensembles (potentiellement vides) de contraintes sur motifs, et $<$ est un pré-ordre strict [3] sur les motifs vérifiant $\Pi < \Pi' \Rightarrow inc_{R^-}(\Pi) \leqslant inc_{R^-}(\Pi')$. Une solution de P est un motif minimal pour $<$ qui satisfait toutes les contraintes de $R^+ \cup R^-$ sur D^+.*

Notons qu'un MDP tolère l'utilisation de critères d'optimisation secondaires. Nous présentons ci-dessous deux mesures - cardinalité et variabilité de motif - qui peuvent être combinées avec la mesure d'inclusion. La variabilité d'un motif est le nombre maximum de hashs consécutifs des patrons de ses c-blocs.

Définition 10 (Mesures) *Soit Π un motif, $card(\Pi) = |\Pi|$ est la cardinalité de Π, et $slack(\Pi) = \max_{(p,\delta)\in\Pi, i\in[|\pi(p)|-1]} (\pi(p)(i+1) - \pi(p)(i) - 1))$ est la variabilité de Π.*

3. Un pré-ordre strict est un ordre partiel strict pour lequel la relation d'incomparabilité associée est transitive.

Nous définissons ci-dessous la classe RMDP (Replication-free MDP) de MDP dont les motifs solutions sont fermés sur la classe positive, sans-réplications et couvrant minimalement la classe négative.

Définition 11 (RMDP) *RMDP est la classe de MDP telle que* $R^+ = \{\texttt{fermeture}, \texttt{non-réplication}\}$ *et* $R^- = \{\texttt{couverture}\}$.

De par les propriétés de la contrainte de non-réplication, notons que tout RMDP $(D, \{\texttt{fermeture}, \texttt{non-réplication}\}, \{\texttt{couverture}\}, <)$ a mêmes solutions que le MDP $(D, \{\texttt{fermeture}\}, \{\texttt{couverture}, \texttt{non-réplication}\}, <)$. Le résultat suivant établit sous certaines conditions sur la fonction objectif que tout RMDP satisfaisable admet une solution dont les c-blocs sont maximaux pour la relation de réplication. On note $\Pi \lesssim \Pi' \Leftrightarrow (\neg(\Pi < \Pi') \Rightarrow \neg(\Pi' < \Pi))$.

Théorème 1 *Soit* $P = (D, R^+, R^-, <)$ *un RMDP,* C *l'ensemble des c-blocs de* P *et* $C_{\geq_{top}}$ *l'ensemble des c-blocs maximaux pour la réplication. Soit* $f : C \rightarrow C_{\geq_{top}}$ *telle que* $f(c) \geq (c)$ *pour tout* $c \in C$ *et* $F : 2^C \rightarrow 2^{C_{\geq_{top}}}$ *telle que* $F(\{c_1, \ldots, c_k\}) = \{f(c_1), \ldots, f(c_k)\}$. *Si* $F(\Pi) \lesssim \Pi$ *pour tout* $\Pi \in 2^C$ *et* P *est satisfaisable alors* P *a une solution dans* $2^{C_{\geq_{top}}}$.

La condition du théorème est vérifiée si $<$ est le préordre déterminé par la mesure d'inclusion, c'est-à-dire, $\Pi < \Pi' \Leftrightarrow inc_{R^-}(\Pi) < inc_{R^-}(\Pi')$. Elle est aussi vérifiée si la cardinalité minimum est le second critère, c'est-à-dire, $\Pi < \Pi' \Leftrightarrow (inc_{R^-}(\Pi) < inc_{R^-}(\Pi') \vee (inc_{R^-}(\Pi) = inc_{R^-}(\Pi') \wedge card(\Pi) < card(\Pi')))$. Il en est de même si l'on recherche à maximiser la variabilité de motifs. Dans ces cas de figure, le résultat indique qu'on peut se limiter au calcul des c-blocs maximaux par réplication pour obtenir un motif solution. En outre, les c-blocs maximaux par réplication étant nécéssairement fermés sur la classe positive, la contrainte de fermeture peut être ignorée.

D'un point de vue opérationnel, ce résultat motive une approche en deux étapes qui consiste d'abord à calculer les c-blocs maximaux puis à déterminer un motif solution par optimisation combinatoire. La section suivante présente une approche semblable pour le cas général des RMDP où la fonction objectif ne satisfait pas nécessairement aux conditions du théorème.

3 Méthode de résolution RMDP par décomposition

Cette section présente une méthode pour résoudre tout RMDP dont la fonction objectif combine les critères suivants par ordre de priorité : mesure d'exclusion maximum, variabilité minimale et cardinalité minimale. On impose aussi une borne maximum sur la variabilité de motif solution. Une solution est donc un motif fermé sur la classe positive, sans-réplications et minimal pour la fonction objectif. Un motif solution exclut donc le plus grand nombre possible de séquences négatives.

La méthode procède en deux étapes. La première consiste à calculer un ensemble de c-blocs fermés sur la classe positive et qui est sans-inclusions. La seconde identifie pour chacune des séquences négatives lesquels de ces c-blocs ne la couvrent pas, puis détermine un motif solution.

3.1 Calcul de c-blocs

L'algorithme 1 calcule en trois étapes un ensemble sans-inclusions de c-blocs fermés :

1. La première étape (lignes 1–2) est effectuée par l'algorithme 2. Cet algorithme calcule pour la plus petite séquence positive l'ensemble des blocs qui sont de taille 2 (à deux caractères solides), de variabilité inférieure au maximum autorisé $maxslack$, et dont les patrons couvrent la classe positive.
2. La seconde étape (ligne 3) est effectuée par l'algorithme 3. Cet algorithme calcule pour la plus petite séquence positive l'ensemble des blocs dont la variabilité est inférieure au maximum autorisé, dont les patrons couvrent la classe positive, et qui sont maximaux pour l'inclusion dans cet ensemble. Ce calcul est effectué à partir des blocs précédemment calculés.
3. La dernière étape (ligne 4) est effectuée par l'algorithme 4. Cet algorithme calcule à partir des blocs précédemment calculés un ensemble sans-inclusions de c-blocs fermés.

Algorithme 1
`ComputeClosedCBlocks(D+,maxSlack)`

Require: D^+ est la classe positive ; `maxSlack` est la variabilité maximum autorisée.
1: $s \leftarrow \arg\, min_{t \in D^+} |t|$
2: `minbs` ← `FindAllMinimalBlocks`(D^+, `s`, `maxSlack`)
3: `maxbs` ← `FindAllMaximalBlocks`(D^+, `s`, `maxSlack`, `minbs`)
4: `maxcbs` ← `FindMaximalCBlocks`(D^+, `s`, `maxbs`)
5: **return** `maxcbs`

L'algorithme 2 génère pour la plus petite séquence positive s l'ensemble `minbs` des blocs minimaux dont la couverture est D^+. Pour chaque valeur de variabilité autorisée et chacun des blocs de s dont le patron vérifie $|\pi(p)| = 2$ (ligne 3), l'algorithme vérifie si la couverture de p est D^+ (ligne 4). Lorsqu'un tel bloc est détecté, l'ensemble `minbs` est mis à jour (ligne 5). Soient $n = |D^+|$, $\underline{d}$ et $\overline{d}$ respectivement les longueurs minimum et maximum des séquences positives. La variabilité d'un bloc est bornée par $\underline{d} - 2$ car la séquence s est la plus petite dans D^+. De plus, le nombre maximum de blocs minimaux pour n'importe quelle valeur

Algorithme 2
`FindAllMinimalBlocks`(D^+,s,`maxSlack`)

Require: D^+ est la classe positive, s est une séquence positive, et `maxSlack` est la variabilité maximum autorisée.
1: `minbs` $\leftarrow \varnothing$
2: **for all** $\sigma \in$ [`maxSlack`] **do**
3: **for all** $b = (p, l, s)$ s.t. $|\pi(p)| = 2 \wedge \sigma = slack(p)$ **do**
4: **if** $\varphi(p) = D^+$ **then**
5: `minbs` $\leftarrow$ `minbs` $\cup \{b\}$
6: **return** `minbs`

Algorithme 3
`FindAllMaximalBlocks`(D^+,s,`maxSlack`,`minbs`)

Require: D^+ est la classe positive, s est une séquence positive, `maxSlack` est la variabilité maximum autorisée, et `minbs` est l'ensemble des blocs minimaux de s à couverture positive.
1: `maxbs` $\leftarrow$ `minbs`
2: **repeat**
3: `oldbs` $\leftarrow$ `maxbs`
4: `newbs` $\leftarrow \varnothing$
5: **for all** $\{b' \leftarrow (p', l', s), b'' \leftarrow (p'', l'', s)\} \subseteq$ `oldbs` **do**
6: {For a block $b = (p, l, s)$ we denote $\mu(b) = \{l + k | k \in \pi(p)\}$}
7: $b \leftarrow (p, min(l', l''), s)$ such that $\mu(b) = \mu(b') \cup \mu(b'')$
8: $\sigma \leftarrow slack(p)$
9: **if** $\sigma \leqslant$ `maxSlack` $\wedge\ b \notin$ `newbs` $\wedge\ b \notin$ `oldbs` **then**
10: **if** $\varphi(p) = D^+$ **then**
11: `newbs` $\leftarrow$ `newbs` $\cup \{b\}$
12: **if** `newbs` $\neq \varnothing$ **then**
13: `maxbs` $\leftarrow$ `newbs`
14: **for all** $b \in$ `oldbs` **do**
15: **if** $\nexists b' \in$ `newbs` such that $b \sqsubseteq b'$ **then**
16: `maxbs` $\leftarrow$ `maxbs` $\cup \{b\}$
17: **until** `newbs` $= \varnothing$
18: **return** `maxbs`

de variabilité est borné par $\underline{d} - 1$. En outre, vérifier que la couverture d'un patron est D^+ est de coût inférieur à $n\,\overline{d}$. La complexité en pire cas de `FindAllMinimalBlocks` est donc $\mathcal{O}(n\,\underline{d}^2\,\overline{d})$.

L'algorithme 3 calcule l'ensemble `maxbs` des blocs de la séquence positive s dont les patrons couvrent la classe positive et qui sont maximaux pour l'inclusion dans cet ensemble. L'idée est de construire cet ensemble incrémentalement par fusion de blocs déjà générés. `maxbs` est initialisé à l'ensemble pré-calculé des blocs minimaux (ligne 1). A chaque itération (lignes 2–17), l'ensemble des blocs déjà calculés, noté `oldbs`, est examiné pour construire les blocs du niveau suivant, noté `newbs`. Initialement, `oldbs` est fixé à `maxbs` et `newbs` à l'ensemble vide (lignes 3–4). Un nouveau bloc b est généré (ligne 7) en fusionnant chaque paire de blocs du niveau courant (ligne 5). Si sa variabilité est inférieure à la valeur maximale autorisée et si ce bloc est nouveau (ligne 9), l'algorithme vérifie alors si son patron est positif (ligne 10). Le cas échéant, le bloc est ajouté à l'ensemble `newbs` (ligne 11). Si au moins un bloc nouveau a été généré (ligne 12), l'algorithme élimine tout bloc de `oldbs` qui n'est plus maximal pour l'inclusion (lignes 13–16).

Le nombre d'itérations réalisées par la boucle de la ligne 2 est borné par $\underline{d}$ car le nombre maximum de caractères solides des blocs de `newbs` ne peut que croître à chaque itération et $\underline{d}$ est la taille de la séquence s sur

Algorithme 4 `FindMaximalCBlocks`(D^+,s,`maxbs`)

Require: D^+ est la classe positive, s est une séquence positive, et `maxbs` est l'ensemble des blocs de s à couverture positive et maximaux pour l'inclusion.
1: `maxcbs` $\leftarrow \varnothing$
2: **for all** $(p, l, s) \in$ `maxbs` **do**
3: $\forall t \in D^+\ \ \delta(t) \leftarrow \varnothing$
4: $\delta(s) \leftarrow l$
5: `cbFound` $\leftarrow$ `TRUE`
6: **for all** $t \in D^+$ such that $t \neq s$ **do**
7: **if** $\exists (p, l', t)$ such that
$\forall (p', \delta') \in$ `maxcbs` $(p, l', t) \not\sqsubseteq (p', \delta'(t), t)$ **then**
8: $\delta(t) \leftarrow l'$
9: **else**
10: `cbFound` $\leftarrow$ `FALSE`
11: **if** `cbFound` **then**
12: `maxcbs` $\leftarrow$ `maxcbs` $\cup \{(p, \delta)\}$
13: **return** `maxcbs`

laquelle nous calculons les blocs maximaux. Soit m le nombre maximum de blocs trouvés lors d'une itération. Le nombre d'itérations réalisées par la boucle de la ligne 5 est borné par m^2. Vérifier si un bloc existe déjà est de coût inférieur à m et vérifier si un bloc a une couverture positive est de coût inférieur à $n\overline{d}$. La complexité en pire cas de `FindAllMaximalBlocks` est donc $\mathcal{O}(\underline{d}\,m^2(m + n\,\overline{d}))$.

L'algorithme 4 calcule un ensemble sans-inclusions `maxcbs` de c-blocs fermés sur la classe positive. Le nombre maximum de c-blocs est borné par |`maxbs`| car chacun des blocs $b \in$ `maxbs` de la séquence s peut-être associé à un c-bloc au plus. Pour chaque bloc (p, l, s) précalculé (ligne 2), l'algorithme tente de trouver un c-bloc (lignes 3–12). Il crée d'abord une fonction d'enracinement δ nulle (ligne 3), puis fixe la valeur de $\delta(s)$ à la position l (ligne 4). Il essaie ensuite pour chaque séquence positive t de trouver un bloc dont le patron est p et qui n'est pas inclus dans les blocs de c-blocs précédemment obtenus (lignes 6–7). Si l'algorithme parvient à définir totalement δ alors le nouveau c-bloc est ajouté à l'ensemble `maxcbs`. Soit $m =$ |`maxbs`|, le coût pour trouver un bloc maximal dans chacune des séquences est de $\overline{d}$, et le coût pour vérifier si un bloc a déjà été découvert est de m. La complexité en pire cas de `FindMaximalCBlocks` est donc $\mathcal{O}(mn\,(\overline{d} + m))$.

3.2 Calcul de motif optimal

Une fois calculé l'ensemble sans-inclusions de c-blocs fermés, la seconde étape de la méthode de résolution RMDP détermine un motif optimal. Cette étape se ramène à la résolution d'un problème de couverture minimum par ensembles. Nous en présentons une modélisation sous la forme d'un problème d'optimisation sous contraintes.

Variables et Contraintes. Soit C l'ensemble des c-blocs calculés. Pour chaque séquence négative $s \in D^-$, nous calculons le sous-ensemble des c-blocs de C qui ne couvrent pas s. Cet ensemble est noté $E_s \subseteq C$. Nous associons à chaque c-bloc $i \in C$ une variable $x_i \in \{0, 1\}$

Tableau 1 – Résultats obtenus pour le jeu de séquences protéiques LEAP par résolution RMDP.

cid	#proteins	$\underline{d}$	$\overline{d}$	#nonexp	card	slack	length	time1	time2
1	177	117	507	35	13	14	29	66775	237
2	96	122	338	3	9	27	25	276043	402
3	29	86	186	0	1	0	6	92	124
4	83	81	625	690	1	3	2	6745	8
5	60	83	217	0	3	2	8	311	121
6	202	66	843	258	3	6	6	4079	18
7	53	95	341	0	1	4	4	127	23
8	184	136	411	84	2	1	4	7016	17
9	67	78	144	0	1	2	4	45	15
10	76	88	173	2	7	46	18	49962	674
11	24	159	278	0	5	1	13	358	292
12	15	71	117	0	2	0	6	55	57

indiquant si i appartient au motif ($x_i = 1$) ou non ($x_i = 0$).

Pour chaque séquence négative $s \in D^-$, on souhaite sélectionner au moins un c-bloc qui ne couvre pas s s'il en existe. La mesure d'inclusion du motif est donnée par $inc = \sum_{s \in D^-} (\sum_{i \in E_s} x_i = 0)$. La variabilité du motif est égale à la variabilité maximum de ses c-blocs, c'est-à-dire $slack = max_{i \in C}(slack(i).x_i)$ où $slack(i)$ dénote la variabilité du c-bloc i. La cardinalité du motif est donnée par $card = \sum_{i \in C} x_i$.

Objectif. L'objectif est de minimiser par ordre de priorité l'inclusion, la variabilité et la cardinalité du motif.

$$\texttt{minimize}\ card + \alpha \cdot slack + \beta \cdot inc$$

α et β sont des coefficients dont la valeur est fixée de manière à respecter l'ordre entre les trois critères.

4 Résultats empiriques

La méthode présentée ci-dessus calcule un ensemble de c-blocs pour la variabilité maximum autorisée puis un motif solution. Or il n'est pas toujours nécessaire de calculer tous ces c-blocs si l'on parvient à exclure toutes les séquences négatives à moindre variabilité. L'idée est donc d'utiliser une approche paresseuse qui consiste à incrémenter la valeur de variabilité autorisée de 0 jusqu'à sa valeur maximale ($maxSlack$) en appliquant la méthode à chaque pas. Cette boucle s'arrête dès qu'un motif solution parvient à exclure toutes les séquences négatives. Les résultats présentées ci-dessous font référence à cette approche. A noter que l'implémentation actuelle redémarre la résolution depuis le début à chaque itération. On peut néanmoins envisager de rendre l'algorithme plus incrémental.

Nous présentons ci-dessous les résultats de cette approche sur deux bases de séquences protéiques prépartitionnées : Late Embryogenesis Abundant Proteins (LEAP) et Small Heat Shock Proteins (SHSP). La base LEAP [7] contient 1066 séquences réparties en 12 classes tandis que la base SHSP [8] contient 2244 séquences réparties en 23 classes. Les algorithmes ont été programmés en Java. Les expériences ont été réalisées sous distribution Linux et architecture équipée d'un processeur quatre-coeurs de 2.66 GHz avec 3.8 GB de RAM. Tous les tests ont été menés jusqu'au terme.

Les résultats sont détaillés dans les tableaux 1 et 2. Chacune des classes a été traitée tour à tour comme étant la classe positive, les séquences des autres classes formant alors la classe négative. `cid` correspond à l'identifiant de la classe traitée, `#proteins` correspond au nombre de ses séquences, $\underline{d}$ à leur taille minimum et $\overline{d}$ à leur taille maximum. Pour 6 des 12 classes de LEAP et 3 des 23 classes de SHSP, nous n'avons pas trouvé de motif qui excluait la totalité des séquences négatives. La colonne `#nonexp` indique le nombre de séquences négatives qui n'ont pu être exclues par le motif solution pour chacune des classes. A noter qu'aucune séquence négative n'a pu être exclue pour la classe 11 de SHSP. Les mesures de variabilité, cardinalité et longueur[4] des motifs solutions sont données respectivement dans les colonnes `slack`, `card`, et `length`. Les colonnes `time1` et `time2` donnent le temps de calcul en millisecondes utilisés respectivement pour le calcul de c-blocs et pour le calcul de motif optimal.

Notons que nous avons aussi modélisé le problème RMDP en utilisant Minizinc [10]. Toutefois, les modèles obtenus n'ont pu être appliqués qu'à de petites instances de par leur taille réhdibitoire. Les outils comme Miningzinc (CP for Data mining) ont également été considérés mais ils ne proposaient aucune fonctionnalité pour traiter les données séquentielles.

Un RMPD n'impose aucun séquencement sur les c-blocs d'un motif solution. Cette flexibilité peut se révéler utile comme en témoigne le tableau 3. Ce tableau donne l'enracinement d'un motif solution constitué de 4 c-blocs pour

4. La longueur d'un motif est la somme des nombres de caractères solides des c-blocs qui le composent.

Tableau 2 – Résultats obtenus pour le jeu de séquences protéiques SHSP par résolution RMDP.

cid	#proteins	$\underline{d}$	$\overline{d}$	#nonexp	card	slack	length	time1	time2
1	237	130	163	0	10	10	20	4122	511
2	107	129	174	0	3	2	7	760	258
3	47	165	328	0	5	3	13	1215	388
4	16	119	173	0	3	0	10	251	226
5	14	172	203	0	2	0	6	356	275
6	65	127	248	0	7	8	15	3569	531
7	80	163	266	0	4	3	9	1112	412
8	15	115	146	0	6	1	14	597	357
9	146	121	170	0	4	2	10	236	437
10	294	120	498	1747	1	1	2	5471	9
11	295	130	316	1949	-	-	-	4815	0
12	257	149	269	0	8	25	16	3800	274
13	119	174	271	0	1	1	4	420	332
14	25	145	178	0	1	0	5	473	467
15	25	108	262	0	1	0	7	492	554
16	31	114	154	0	1	0	4	215	257
17	69	102	131	0	3	0	7	96	104
18	154	107	277	77	1	0	2	7765	36
19	23	194	220	0	4	1	9	665	483
20	90	214	298	0	9	3	18	656	251
21	96	161	344	0	3	1	6	217	149
22	13	332	453	0	2	0	7	1474	1392
23	26	79	127	0	2	1	5	342	161

Tableau 3 – Un motif optimal pour la 19^{me} classe de la base SHSP. La cardinalité du motif vaut 4, sa variabilité 1 et sa longueur 9.

protein	PE#V	D#K	Q#S	CS
1	149	90	8	159
2	140	91	8	3
3	140	91	8	3
4	140	91	8	3
5	140	91	8	3
6	147	98	8	157
7	140	91	8	3
8	140	91	8	3
9	147	98	8	157
10	144	75	8	3
11	151	127	69	156
12	137	83	157	142
13	144	75	8	154
14	152	98	8	157
15	141	87	8	146
16	148	89	8	17
17	150	71	8	17
18	140	91	8	3
19	141	122	189	151
20	141	122	189	151
21	140	91	8	3
22	134	85	172	144
23	141	92	81	151

une des classes SHSP. Chaque ligne correspond à l'une des séquences positives et chaque colonne donne la position de départ d'un des c-blocs dans cette séquence. On remarque que les c-blocs ne suivent pas le même ordre selon les séquences.

5 Conclusion

Nous avons présenté le formalisme MDP pour le calcul de motifs discriminants sur données séquentielles prépartitionnées. Ce problème revêt un intérêt particulier en Bio-informatique pour valider a posteriori la classification experte de séquences protéiques en signant chaque classe par un motif exclusif. Le formalisme MDP repose sur un modèle de motifs à base de c-blocs qui permet de distinguer contraintes sur c-blocs et contraintes entre c-blocs. Il permet en outre de limiter le nombre de contraintes à traiter selon leurs propriétés (anti-monotonie, subsomption, etc). Dans ce cadre, nous avons présenté plusieurs contraintes visant à éliminer des motifs redondants (fermeture, non-inclusion) ou à satisfaire des besoins applicatifs (non-réplication, séquentialité).

Nous nous sommes intéressés à la classe RMDP où l'on recherche des motifs fermés sur la classe positive, sans-réplications et couvrant minimalement la classe négative. Sous certaines conditions portant sur la fonction objectif, nous avons établi qu'il suffisait de se limiter au calcul de c-blocs maximaux par réplication afin d'obtenir un motif solution. Dans le cas général, nous avons proposé une mé-

thode en deux étapes consistant à construire des c-blocs fermés puis à résoudre un problème de couverture par ensembles. L'efficacité de cette méthode a été démontrée sur des bases de séquences protéiques.

Nous entendons poursuivre ce travail dans diverses directions : mise en oeuvre de nouvelles contraintes sur motifs, étude de la complexité théorique de différentes classes MDP selon le langage de contraintes, correspondance avec le problème d'alignement de séquences et comparaison expérimentale avec des algorithmes dédiés, et extension au problème de partitionnement automatique (clustering).

Références

[1] Emmanuel Coquery, Said Jabbour, Lakhdar Sais, and Yakoub Salhi. A SAT-Based approach for discovering frequent, closed and maximal patterns in a sequence. In *ECAI*, pages 258–263, 2012.

[2] Emmanuel Coquery, Said Jabbour, and Lakhdar Saïs. A constraint programming approach for enumerating motifs in a sequence. In *2011 IEEE 11th International Conference on Data Mining Workshops (ICDMW)*, pages 1091–1097, 2011.

[3] Tias Guns, Siegfried Nijssen, and Luc De Raedt. Itemset mining : A constraint programming perspective. *Artificial Intelligence*, 175(12–13) :1951–1983, August 2011.

[4] Tias Guns, Siegfried Nijssen, and Luc De Raedt. k-pattern set mining under constraints. *IEEE Transactions on Knowledge and Data Engineering*, 25(2) :402–418, 2013.

[5] Dan Gusfield. *Algorithms on strings, trees, and sequences : computer science and computational biology*. Cambridge University Press, Cambridge [England] ; New York, 2008.

[6] Gilles Hunault and Emmanuel Jaspard. LEAPdb : a database for the late embryogenesis abundant proteins. *BMC Genomics*, 11(1) :221, April 2010. PMID : 20359361.

[7] Gilles Hunault and Emmanuel Jaspard. The late embryogenesis abundant proteins DataBase. `http://forge.info.univ-angers.fr/~gh/Leadb/index.php`, 2013.

[8] Gilles Hunault and Emmanuel Jaspard. The small heat shock proteins database. sHSPdb. `http://forge.info.univ-angers.fr/~gh/Shspdb/index.php`, 2013.

[9] Jean-Philippe Métivier, Samir Loudni, and Thierry Charnois. A constraint programming approach for mining sequential patterns in a sequence database.

[10] Nicholas Nethercote, Peter J Stuckey, Ralph Becket, Sebastian Brand, Gregory J Duck, and Guido Tack. Minizinc : Towards a standard cp modelling language. In *Principles and Practice of Constraint Programming–CP 2007*, pages 529–543. Springer, 2007.

Calepinage à base de contraintes : application à la rénovation de bâtiments à haute performance énergétique

A.F. Barco[1] **É. Vareilles**[1] **M. Aldanondo**[1] **P. Gaborit**[1] **M. Falcon**[2]

[1] Université de Toulouse - Mines Albi - France
[2] TBC Générateur d'Innovation - Colomiers - France
{abarcosa, vareille, aldanond, gaborit}@mines-albi.fr mfalcon@tbcinnovation.fr

Résumé

Nous présentons dans cette communication une première version d'un algorithme à base de contraintes dédié à la génération de solutions de calepinage de façades. Cet algorithme permet de paver une façade de panneaux multifonctionnels rectangulaires et de dimensions variables et pouvant porter plusieurs équipements (menuiseries, panneaux solaires, gouttières). Plusieurs contraintes doivent être prises en compte pour converger vers une solution : des contraintes de fabrication, d'accessibilité, de transport, de couverture, de poids et de fixation des panneaux en façade. Le détail de l'algorithme et le modèle à base de contraintes supportant cette problématique de calepinage forment le coeur de cet article. Nos propositions sont illustrées sur un exemple réel issu du chantier démonstrateur du projet C.R.I.B.A.

Abstract

We present in this communication the first version of a contraint-based algorithm dedicated to the layout design of building facades. This algorithm is able to tile a facade with multi-functional and rectangular panels with varying dimensions and containing eventuelly a set of equipments (windows, solar modules, rain gutters, etc). Different kinds of contraints have to be satisfied in order to reach a solution : manufacturing constraint, accessibility constraint, supply constraint, covering constraint, weight constraint and fixation constraint. Details of the layout design algorithm and its underlying constraints model are the core of this paper. We illustrate our proposals on a real example coming from the pilote building site of the C.R.I.B.A. project.

1 Introduction

La rénovation énergétique est l'un des enjeux majeurs de notre siècle. La consommation énergétique des bâtiments résidentiels et commerciaux ne cesse de croître et dépasse actuellement les secteurs de l'industrie et des transports. Elle représente plus du tiers de la consommation d'énergie dans les pays développés : 44% en France [1], 37% en Europe [10], 36% en Amérique du Nord [3] et 31% au Japon [2]. L'augmentation de la population, l'engouement pour les nouvelles technologies et l'amélioration du confort de vie combinés à la modification des habitudes domestiques font que la demande en énergie des bâtiments va inlassablement continuer d'augmenter dans les prochaines années. Par conséquent, réduire la consommation en énergie des bâtiments est maintenant un objectif prioritaire à la fois national et international.

Plusieurs pays ont adhéré au protocole de Kyoto et ont mis en place des politiques de réduction de la consommation énergétique des bâtiments. Ces dernières se sont tout d'abord concentrées sur les constructions neuves. Mais vu le faible taux de constructions neuves en Europe (1 à 2 %[11]) et de la durée de vie des bâtiments, les efforts portent maintenant sur la rénovation du parc de bâtiments existants. En France, cet effort sur l'existant a été confirmé par la loi Grenelle 1 de 2009 qui fixe comme objectif « de réduire les consommations d'énergie du parc des bâtiments existants d'au moins 38% d'ici à 2020. » Pour ce faire, le gouvernement Français a mis en place en 2013 un plan d'investissement pour atteindre l'objectif

1. http://www.developpement-durable.gouv.fr/IMG/pdf/Rep_-_chiffres_energie.pdf

de 500 000 logements rénovés par an à l'horizon 2017, dont 120 000 logements sociaux [2]. Il est donc primordial d'amorcer cette rénovation massive de bâtiments et de l'assister par des systèmes d'aide à la décision [6].

Nos travaux s'inscrivent dans le cadre du projet C.R.I.B.A. pour Construction et Rénovation Industrialisées en Bois Acier[5]. Ce projet porte sur l'industrialisation de la rénovation énergétique de bâtiments collectifs. La performance énergétique atteinte par le bâtiment après rénovation doit être moins de $25kWh/m^2/an$. Pour ce faire, le bâtiment est totalement recouvert d'une nouvelle enveloppe composée de panneaux multifonctionnels, rectangulaires et préfabriqués en usine.

Nous présentons dans cette communication la première version d'un algorithme de calepinage à base de contraintes permettant de définir une solution de calepinage de façade. Dans un premier temps, en section 2, nous exposons et définissons la problématique de calepinage de façades. Dans un deuxième temps, en section 3, nous décrivons le modèle théorique de connaissances formalisé comme un CSP supportant cette problématique de calepinage. Puis, en section 4, nous présentons la première version de l'algorithme de calepinage à base de contraintes. Enfin, en section 5, nous illustrons son fonctionnement sur un exemple réel issu du chantier de rénovation démonstrateur du projet C.R.I.B.A. Nous devons souligner que les travaux présentés dans cet article restent à ce jour conceptuels et théoriques. Le modèle de connaissances ainsi que l'algorithme de calepinage sont en cours d'implémentation sur CHOCO (`http://www.emn.fr/z-info/choco-solver/`).

2 Calepinage de façades

Afin d'obtenir la performance énergétique cible du projet C.R.I.B.A. et garantir l'étanchéité à l'air du bâtiment, chaque façade du bâtiment rénové doit être entièrement recouverte de panneaux multifonctionnels. Une solution de calepinage d'une façade revient donc à trouver un ensemble de panneaux multifonctionnels positionnés dans l'espace et couvrant la totalité de la façade, sans superposition, ni trou. Nous pouvons, par conséquent, apparenter le calepinage de façades au pavage de celles-ci. En effet, un pavage (ou dallage) est une partition d'un espace (ici les façades) par des éléments d'un ensemble fini, appelés tuiles (ici les panneaux multifonctionnels).

Cependant, dans notre cas d'application, la forme géométrique des tuiles est certes définie (forme rectangulaire) mais d'une part, leur surface, leurs dimensions et leur poids peuvent varier au fil de la résolution, et d'autre part, leur nombre n'est *a priori* pas connu en avance.

2. `http://www.gouvernement.fr/sites/default/files/fichiers_joints/dossier_de_presse_du_ministere_de_legalite_des_territoires_et_du_logement.pdf`

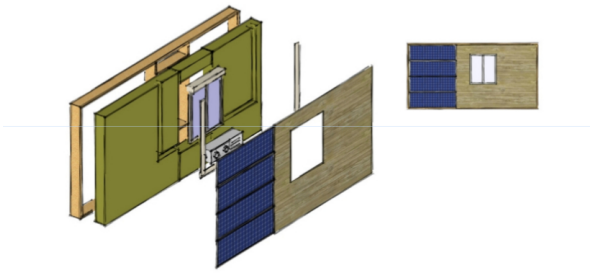


FIGURE 1 – Panneau multifonctionnel

2.1 Éléments de calepinage

Deux éléments sont principalement à considérer pour définir le calepinage :

- les façades : nous définissons une façade comme un plan vertical positionné dans l'espace. Ce plan contient uniquement des zones rectangulaires définissant :
  - la façade à rénover avec ses dimensions (hauteur et largeur),
  - des menuiseries avec leurs dimensions (hauteur, largeur) positionnées dans le plan de référence,
  - les zones de fixation avec leur charge admissible positionnées dans le plan de référence,
  - les zones qualifiées de « hors configuration » qui ne peuvent être couvertes par des panneaux rectangulaires multifonctionnels industrialisés et nécessitent par conséquent une conception spécifique (panneaux triangulaires, trapézoïdaux, etc).
- les panneaux multifonctionnels (présentés en figure 1) : les panneaux multifonctionnels sont rectangulaires, de dimensions (de 0.9 à 13.5 m) et surface variables (de 1 à $45.5m^2$) et peuvent inclure différents équipements (menuiseries, panneaux photovoltaïques, etc). Ces panneaux sont dimensionnés un à un lors de la définition du calepinage, puis préfabriqués en usine avant transport et montage sur le chantier de rénovation.

La précision demandée pour la préfabrication des panneaux rectangulaires en usine (dimensions et position des menuiseries) avant leur transfert et leur montage sur chantier est de l'ordre du millimètre.

2.2 Processus de calepinage

Le processus de rénovation se déroule suivant un *travelling optique avant* à plusieurs niveaux partant du chantier et arrivant aux panneaux multifonctionnels [13]. À chacun des niveaux, une série de questions descriptives est posée à l'utilisateur. Chaque réponse a un impact potentiel sur la surface, les dimensions et le poids admissibles des panneaux multifonctionnels. Par exemple, l'inaccessibilité d'une façade limite la taille des panneaux multifonctionnels et, par conséquent, la surface couverte par chacun d'eux.

Une fois les descriptions du chantier, du bâtiment et des façades terminées, le calepinage de chaque façade peut commencer. Les façades doivent porter un ensemble minimal de panneaux qui doivent être les plus grands possibles tout en respectant les contraintes architecturales, de fixation en façade, de fabrication et d'accessibilité. Un panneau multifonctionnel est adéquat s'il respecte les conditions suivantes :

C1 il doit être aussi couvrant que possible au regard des contraintes d'accessibilité et de la position géométrique des menuiseries,

C2 il peut être posé en façade et porté par une ou plusieurs zones de fixation,

C3 il ne se superpose avec aucun autre panneau,

C4 il n'interfère pas avec la définition et le placement des autres panneaux.

Le calepinage d'une façade est qualifié de solution si chaque panneau multifonctionnel le composant est adéquat et si l'ensemble des panneaux forme bien une partition de la façade considérée.

2.3 Exemple de calepinage

Considérons la façade à rénover, notée (a), de la figure 2. Les lignes horizontales et verticales représentent les lignes de fixation autorisées. Elles correspondent aux différents emplacements possibles pour les fixations supportant le poids des panneaux. Nous considérons dans cet article que celles-ci sont capables de supporter un poids suffisamment grand pour ne pas contraindre la surface et les dimensions des panneaux.

Les fixations sont constituées de deux parties : une partie fixée directement sur la façade à rénover (fixation murale) et une partie posée en usine sur les panneaux. Au niveau des façades, les fixations sont positionnées au centre des lignes de fixation. Au niveau des panneaux, nous nous plaçons dans le cas où les panneaux sont posés sur leurs fixations (et non pas suspendus ou agrafés à celles-ci). Les fixations sont donc positionnées en usine sur le bord inférieur des panneaux à équidistance (2 mètres au minimum) les unes des autres. Cette distance minimale de 2 mètres permet de répartir au mieux le poids des panneaux supportés. Une fixation murale peut supporter un unique panneau (si elle se trouve sur le périmètre de ce dernier) ou deux panneaux (si celle-ci se trouve à l'angle du panneau considéré). Un seul mode de fixation (posé, suspendu ou agrafé) n'est autorisé par façade rénovée.

Les petits rectangles présents sur la façade à rénover correspondent aux emplacements des menuiseries (portes et fenêtres).

Deux zones de la façade sont considérés comme « hors configuration » : le pignon et le pied d'immeuble. Deux panneaux spécifiques seront conçus, respectivement triangulaire pour le pignon et avec un revêtement extérieur spécifique imputrescible pour le pied d'immeuble.

La façade, notée (b), de la figure 2, présente trois panneaux inadéquats :

- le panneau P_1 ne respecte pas la condition *C4* car il empêche de fixer un panneau multifonctionnel couvrant la zone au-dessus de lui. Il déborde en effet sur la ligne de fixation positionnée à 12 mètres de haut, empêchant tout autre panneau d'être positionné à cet endroit là. Cette zone ne pourra donc pas être couverte par la nouvelle enveloppe et par conséquent, aucune solution de calepinage ne pourra être trouvée.
- Le panneau P_2 ne peut pas être fixé en façade et ne respecte pas la condition *C2*. Seul son coté droit est aligné sur une zone de fixation verticale. Il ne peut pas à lui seul supporter la totalité du poids du panneau.
- Le panneau P_3 ne respecte pas la condition *C1* car il intersecte une menuiserie. Le principe constructif retenu dans le cadre du projet C.R.I.B.A. impose que les menuiseries soient incluses dans un unique panneau.

Les façades notées (c), (d) et (e) de la figure 2 présentent trois solutions de calepinage horizontal (c), vertical (d) et horizontal-vertical (e) où tous les panneaux multifonctionnels respectent les quatre conditions énoncées. La solution à privilégier pour cette façade est la configuration (e) car celle-ci présente le moins de panneaux (5 au lieu de 7 pour la configuration (c) et 6 pour la configuration (d)) et donc le moins de déperdition énergétique (car moins de jonction entre panneaux).

3 Modèle de connaissances

Nous introduisons dans cette section le modèle conceptuel de connaissances, formalisé comme un CSP, supportant cette problématique de calepinage. Un CSP est un triplet $\{X, D, C\}$ avec X l'ensemble des variables, D est l'ensemble des domaines (un do-

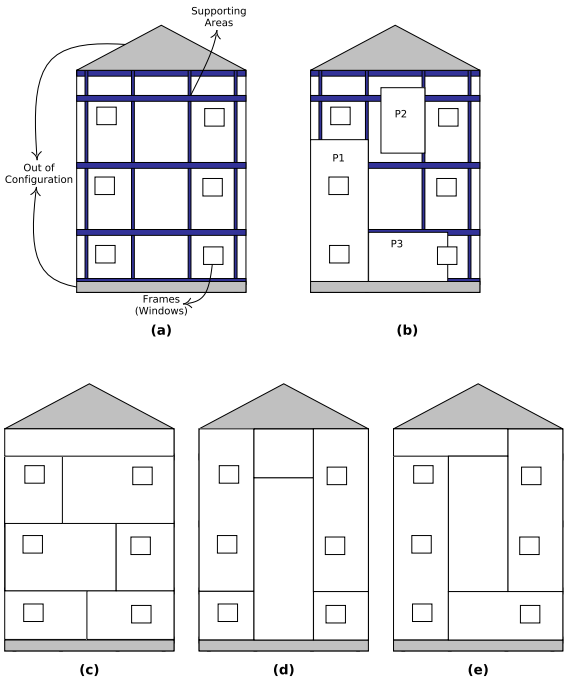


FIGURE 2 – Exemple de calepinage avec panneaux adéquats et non adéquats

maine associé à chacune des variables) et C l'ensemble des contraintes liants les variables [9]. Les variables peuvent être soit discrètes, soit continues. Les contraintes sont des contraintes de compatibilité, lorsqu'elles définissent les combinaisons de valeurs permises ou interdites des valeurs possibles pour l'ensemble des variables ; soit des contraintes d'activation d'un sous-ensemble de variables et de contraintes (DCSP).

3.1 Paramètres et Variables

Nous présentons dans cette sous-section uniquement les 16 paramètres et variables nécessaires à la définition d'une solution de calepinage. La liste des paramètres et variables associées à leur domaine est présentée ci-dessous :

- Paramètres :
  - w_{fac} correspond à la largeur de la façade. Son domaine est continu et compris entre $[0.5, 50]$ mètres,
  - h_{fac} correspond à la hauteur de la façade. Son domaine est continu et compris entre $[2.5, 50]$ mètres,
  - e_{fac} correspond à l'accessibilité de la façade. Son domaine est symbolique et ne possède que deux valeurs $\{accessible, \overline{accessible}\}$,
  - (f_{x0}, f_{y0}) correspond au couple de coordonnées de l'angle inférieur gauche d'une menuiserie. Le domaine des variables est continu et compris respectivement entre $[0, w_{fac}]$ pour l'abscisse et entre $[0, h_{fac}]$ pour l'ordonnée,
  - (f_{x1}, f_{y1}) correspond au couple de coordonnées de l'angle supérieur droit d'une menuiserie. Le domaine des variables est continu et compris respectivement entre $[0, w_{fac}]$ pour l'abscisse et entre $[0, h_{fac}]$ pour l'ordonnée,
  - (hc_{x0}, hc_{y0}) correspond au couple de coordonnées de l'angle inférieur gauche d'une zone hors configuration. Le domaine des variables est continu et compris respectivement entre $[0, w_{fac}]$ pour l'abscisse et entre $[0, h_{fac}]$ pour l'ordonnée,
  - (hc_{x1}, hc_{y1}) correspond au couple de coordonnées de l'angle supérieur droit d'une zone hors configuration. Le domaine des variables est continu et compris respectivement entre $[0, w_{fac}]$ pour l'abscisse et entre $[0, h_{fac}]$ pour l'ordonnée,
  - w_{hc} correspond à la largeur d'une zone hors configuration. Son domaine est continu et compris entre $[0, h_{fac}]$ mètres,
  - h_{hc} correspond à la hauteur d'une zone hors configuration. Son domaine est continu et compris entre $[0, h_{fac}]$ mètres,
  - fai_{load} correspond à la charge maximale supportée par une fixation. Son domaine est continu et compris entre $[0, 500]$ kg,
  - X_{sa} correspond à l'ensemble des lignes de fixation verticales. Pour chaque ligne de fixation verticale, nous devons connaître sa position. Le domaine associé est continu et compris entre $[0, w_{fac}]$,
  - Y_{sa} correspond à l'ensemble des lignes de fixation horizontales. Pour chaque ligne de fixation horizontale, nous devons connaître sa position. Le domaine associé est continu et compris entre $[0, w_{fac}]$,
- Variables :
  - (p_{x0}, p_{y0}) correspond au couple de coordonnées de l'angle inférieur gauche d'un panneau multifonctionnel. Le domaine des variables est continu et compris respectivement $[0, w_{fac}]$ pour l'abscisse et $[0, h_{fac}]$ pour l'ordonnée,
  - (p_{x1}, p_{y1}) correspond au couple de coordonnées de l'angle supérieur droit d'un panneau multifonctionnel. Le domaine des variables est continu et compris respectivement $[0, w_{fac}]$ pour l'abscisse et $[0, h_{fac}]$ pour l'ordonnée,
  - w_p correspond à la largeur d'un panneau multi-

fonctionnel. Son domaine est continu et compris entre $[0.9, 13]$ mètres,
- h_p correspond à la hauteur d'un panneau multifonctionnel. Son domaine est continu et compris entre $[0.9, 13]$ mètres.

3.2 Contraintes

Dans cette section, nous présentons l'ensemble des contraintes régissant la définition d'un calepinage. Notre modèle étant conceptuel, nous utilisons un CSP disjonctif [1]. Un CSP disjonctif est une disjonction de contraintes atomiques $(<, \leq, >, \geq)$. La forme canonique d'une contrainte disjonctive est exprimée comme $C_i = (d_{i1} \vee d_{i2} \vee \cdots \vee d_{im})$ où chaque d_i est une conjonction de contraintes atomiques $d_j = (c_{j1} \wedge c_{j2} \wedge \cdots \wedge c_{jk})$.

Contrainte liée à l'accessibilité L'accessibilité de la façade à configurer contraint les dimensions maximales des panneaux multifonctionnels. Une façade qui n'est pas accessible nécessitera des panneaux multifonctionnels de plus petite taille qu'une façade accessible. Cette contrainte s'exprime de la manière suivante :

$$((w_p \leq \mathcal{T}_w) \wedge (h_p \leq \mathcal{T}_h) \wedge (e_f = \overline{accessible})) \quad (1)$$

où $\mathcal{T}_w$ et $\mathcal{T}_h$ représentent les dimensions maximales autorisées des panneaux multifonctionnels pour une façade inaccessible.

Contrainte de fabrication et de transport Les panneaux multifonctionnels peuvent couvrir une surface comprise entre $[0.81, 45.5] m^2$. Leurs dimensions maximales sont comprises entre $[0.9, 13]$ mètres pour leur largeur et $[0.9, 13]$ mètres pour leur hauteur. Cependant, il n'est pas possible, pour des raisons de fabrication et de transport, de fabriquer un panneau carré de $6.5 * 6.5$ mètres. La hauteur et la largeur des panneaux sont contraintes l'une par rapport à l'autre de la manière suivante :

$$((w_p \in [0.9, 3.5]) \wedge (h_p \in [0.9, 13])) \vee ((w_p \in [0.9, 13]) \wedge (h_p \in [0.9, 3.5])) \quad (2)$$

Contrainte de couverture Une façade à rénover doit être totalement couverte par les panneaux multifonctionnels et les zones hors configuration. Par conséquent, la contrainte suivante est définie :

$$(w_{fac} * h_{fac}) = \Sigma_{i=1}^{N}(w_{pi} \times h_{pi}) + \Sigma_{j=1}^{k}(w_{hcj} \times h_{hcj}) \quad (3)$$

Contrainte de non recouvrement Pour définir une solution de calepinage, les panneaux multifonctionnels ne doivent pas se recouvrir. Pour chaque paire de panneaux p et q, nous définissons une contrainte de non recouvrement de la manière suivante :

$$(p_{x1} < q_{x0}) \vee (q_{x1} < p_{x0}) \vee (p_{y1} < q_{y0}) \vee (q_{y1} < p_{y0}) \quad (4)$$

Contrainte de poids Une fixation fai^k posée en façade sur une ligne de fixation est caractérisée par ses coordonnées (x, y) et sa charge maximale supportée fai^k_{load}. Soit ATP_k l'ensemble des panneaux supportés par la fixation fai^k et COMPUTEWEIGHT(P, n_f) une fonction évaluant le poids d'un panneau multifonctionnel p connaissant ses dimensions w_p et h_p, son type d'isolant, son revêtement extérieur et le poids de l'ensemble des équipements inclus (menuiseries, panneaux photovoltaïques, etc) si le panneau multifonctionnel p en contient et le nombre n_f de fixations supportant le panneau. La fixation doit être capable de supporter l'ensemble des panneaux multifonctionnels. Cette contrainte de poids est exprimée de la manière suivante :

$$\Sigma_{a=1}^{|ATP_k|} computeWeight(ATP_k[a], n_f) \leq fai^k_{load} \quad (5)$$

Contrainte structurelle Les dimensions des panneaux multifonctionnels sont contraintes par la présence de menuiseries dans leur voisinage. Considérons un couple panneau multifonctionnel p (positionné en $((p_{x0}, p_{y0}), (p_{x1}, p_{y1}))$) et menuiserie f (positionné en $((f_{x0}, f_{y0}), (f_{x1}, f_{y1}))$). Premièrement, soit le panneau p recouvre entièrement la menuiserie f (dans ce cas-là, la menuiserie f est incluse dans le panneau), soit le panneau p se trouve à sa droite, à sa gauche, au-dessus ou en dessous. Deuxièmement, quelle que soit la configuration (menuiserie f incluse ou non dans le panneau p), il existe une distance minimale Δ à respecter (liée à la structure interne des panneaux) entre les bords du panneau p et les bords de la menuiserie f. Cette contrainte structurelle est formalisée par la contrainte disjonctive suivante :

$$\begin{aligned}((&p_{x1} + \Delta \leq f_{x0}) \vee (p_{x0} - \Delta \geq f_{x1}) \\ &\vee (p_{y1} + \Delta \leq f_{y0}) \vee (p_{y0} - \Delta \geq f_{y1})) \\ &\vee ((p_{y0} + \Delta \leq f_{y0}) \vee (p_{y1} - \Delta \geq f_{y1}) \\ &\quad \vee (p_{x0} + \Delta \leq f_{x0}) \vee (p_{x1} - \Delta \geq f_{x1})) \end{aligned} \quad (6)$$

Contrainte de fixation Le bas des panneaux p_{y0} (comportant les fixations portées par le panneau) doit être aligné sur une ligne de fixation horizontale Y_{sa}

afin d'être fixé en façade. Cette contrainte de fixation se traduit par la contrainte suivante :

$$\exists y_{sa} \in Y_{sa}, p_{yo} = y_{sa} \quad (7)$$

4 Algorithme de calepinage

Gardons à l'esprit que dans notre application et, contrairement à d'autres travaux sur le pavage dans le secteur du bâtiment ([14], [7],[12],[4] et [8]), seule la géométrie des tuiles est connue à l'avance. Leurs nombre, dimensions et poids sont déterminés au fil de la résolution.

Une première approche consiste à fixer de manière « arbitraire » un nombre n de panneaux multifonctionnels puis à vérifier s'il existe ou non une solution au regard de ce nombre n. Si aucune solution n'est trouvée, n est alors incrémenté de 1. Le nombre de panneaux composant une solution de calepinage peut être fixé de plusieurs manières.

Tout d'abord, considérons uniquement les dimensions autorisées pour les panneaux : $w_p \in [0.9, 13]$ mètres et $h_p \in [0.9, 13]$ mètres. Le nombre maximum de panneaux positionnables sur la largeur w_{fac} d'une façade est alors de $\frac{w_{fac}}{0.9}$. Le nombre minimum de panneaux positionnables est quant à lui égal à $\frac{w_{fac}}{13}$ si le reste de la division égale 0, ou $\frac{w_{fac}}{13} + 1$ sinon. Soient $[min_v, max_v]$ et $[min_h, max_h]$ les nombres minimaux et maximaux de panneaux positionnables sur la largeur et la hauteur d'une façade. Une première approche combine l'ensemble de ces valeurs pour initier la recherche de solutions et converger soit vers une solution, soit atteindre le nombre de panneaux maximal précédemment calculé.

Une deuxième approche consiste à couvrir chaque menuiserie par un panneau multifonctionnel et à étendre ces derniers pour couvrir la totalité de la façade. Si cela n'est pas possible, d'autres panneaux sont ajoutés afin de définir le calepinage final. L'algorithme supportant cette approche est le suivant :

1. Création d'un nombre de panneaux égal aux nombre de menuiseries présentes sur la façade. Pour chaque couple (panneau p, menuiserie m), filtrage des positions $((p_{x0}, p_{y0}), (p_{x1}, p_{y1}))$ par rapport à celles de la menuiserie associée $((f_{x0}, f_{y0}), (f_{x1}, f_{y1}))$ par la contrainte 6,
2. Application de la contrainte liée à l'accessibilité de la façade après la valuation de la variable e_f par l'utilisateur (telle que définie par la contrainte 1),
3. Application de la contrainte de fabrication et de transport (telle que définie par la contrainte 2),
4. Application de la contrainte de fixation (telle que définie par la contrainte 7),
5. Application de la contrainte de non recouvrement (telle que définie par la contrainte 4),
6. Application de la contrainte de couverture (telle que définie par la contrainte 3). Si cette contrainte est inconsistante, alors le nombre de panneaux utilisé n'est pas suffisant. Il faut en ajouter et revenir à l'étape 1, sinon, il existe plusieurs solutions S_k ne tenant pas compte du poids des panneaux,
7. Sélection d'une solution s_t dans S_k,
8. Application de la contrainte de poids au niveau de la solution retenue (telle de définie par la contrainte 5). Si cette dernière contrainte est satisfaite, une solution de calepinage est trouvée, sinon, il faut choisir une autre solution s_g dans s_k (en revenant à l'étape précédente).

5 Exemple illustratif

Nous présentons dans cette section l'application de l'algorithme présenté en section 4, sur une façade existante issue du chantier démonstrateur *La Pince* à Saint-Paul-les-Dax du projet C.R.I.B.A., présentée en figure 3 :

- La figure (a) présente la façade à rénover composée de 6 menuiseries (rectangles noirs), de 4 lignes de fixation horizontales (lignes noires) et de 4 lignes de fixation verticales (lignes noires).
- La figure (b) présente un certain nombre de positions et de dimensions possibles pour les 6 panneaux couvrant les 6 menuiseries en respectant les contraintes 4, 1 et 2.
- La figure (c) présente ces 6 mêmes panneaux avec leur bas alignés sur les lignes de fixation horizontale (respectant la contrainte 7).
- La figure (d) présente les différentes dimensions des 6 panneaux suivant la contrainte de non recouvrement 4.
- À ce stade, la contrainte de couverture 3 ne peut pas être satisfaite. 3 panneaux sont donc ajoutés progressivement au fil de la résolution.
- La figure (e) présente les différentes surfaces couvertes par l'ensemble des 9 panneaux, selon la contrainte 3.
- La figure (f) présente une solution de calepinage respectant l'ensemble des contraintes dont celle de poids 5.

La solution présentée en (f) est bien cohérente avec l'ensemble des contraintes. Cependant, elle risque fort de ne pas être retenue par l'utilisateur pour des questions d'esthétisme (non alignement des joints de jonction entre panneaux, par exemple).

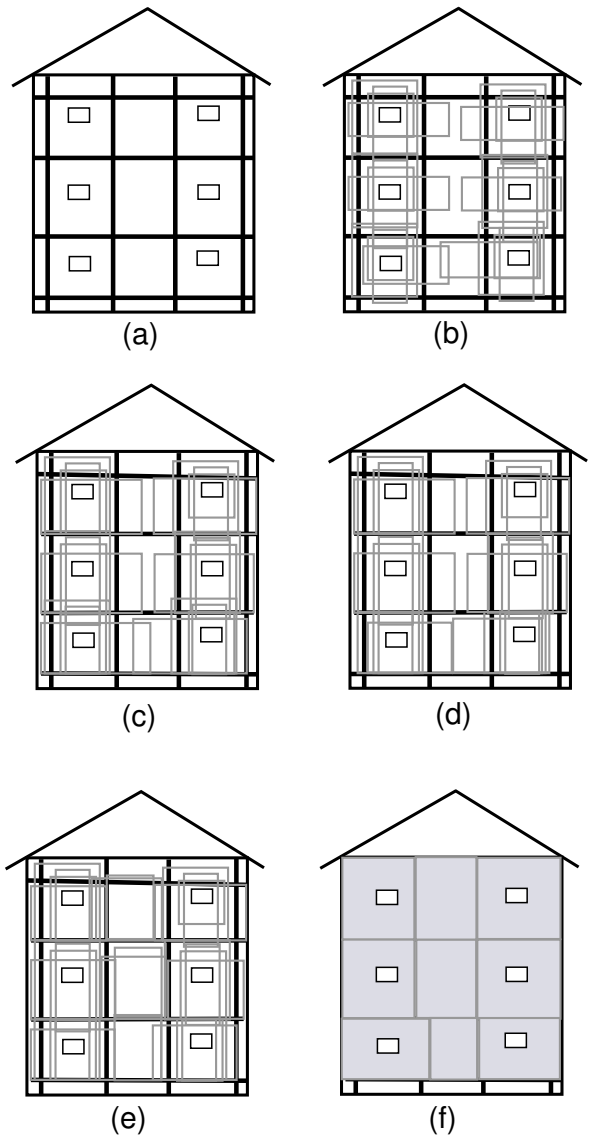


FIGURE 3 – Illustration de l'algorithme de calepinage

6 Conclusion et perspectives

La maîtrise de la consommation énergétique des bâtiments est l'un des défis majeurs du 21^e siècle. La réduction de la consommation énergétique des bâtiments se concentre maintenant sur la rénovation de bâtiments existants. Afin de tenir les objectifs de rénovation fixés par le Gouvernement Français en 2009 et 2013, il est primordial de réaliser cette rénovation de masse avec des outils et des méthodes industrielles plutôt qu'artisanales.

Nous avons présenté dans cette communication un premier algorithme à base de contraintes dédié à la définition du calepinage de façades en vue de leur rénovation. Ces travaux s'inscrivent dans le cadre du projet ADEME C.R.I.B.A. qui vise à industrialiser la rénovation par l'extérieur d'immeubles de logements collectifs pour atteindre une performance énergétique proche de $25kWh/m^2/an$.

Nous avons présenté dans un premier temps notre définition du problème de calepinage de façades en décrivant les spécificités liées à l'isolation des façades par l'extérieur. Dans un deuxième temps, nous avons décrit le modèle de connaissances conceptuel et théorique, supportant cette problématique de calepinage comme un problème de satisfaction de contraintes. L'ensemble des contraintes a été formalisé par des contraintes disjonctives et mathématiques. Celles-ci formalisent, à la fois, des contraintes de fabrication et de transport, mais aussi des contraintes liées à la géométrie et à la structure du bâtiment ainsi qu'à la structure interne des panneaux multifonctionnels. La première version de l'algorithme reprenant l'ensemble de ces contraintes est ensuite présentée et illustrée sur un exemple issu du chantier démonstrateur du projet. Les solutions proposées par notre algorithme sont toutes cohérentes avec l'ensemble des contraintes du problème de calepinage. Ce dernier n'a pas encore été implémenté avec les approches par contraintes.

Cependant, elles ne seront pas toutes retenues par l'utilisateur pour des questions d'esthétisme. Afin d'éviter la génération de solutions non conformes, des connaissances supplémentaires « métier » doivent être ajoutées au modèle de connaissances. Celles-ci sont principalement liées au rendu esthétique du bâtiment après rénovation, comme par exemple une contrainte d'alignement des joints de jonction entre panneaux multifonctionnels ou des contraintes portant sur l'orientation privilégiée des panneaux (verticale ou horizontale).

Les déperditions énergétiques de la façade rapportée se font principalement par les joints de jonction entre panneaux. Actuellement, notre algorithme déduit lors de sa première étape que le nombre de panneaux né-

cessaires à la rénovation est au moins égal au nombre de menuiseries présentes en façade. Il ne permet donc pas de couvrir plusieurs fenêtres avec un même panneau. Une seconde piste d'amélioration concerne donc la minimisation du nombre de panneaux nécessaires à la rénovation et la possibilité de couvrir plusieurs menuiseries avec un seul et même panneau.

Une implémentation poussée avec des outils de résolution de contraintes, tels que CHOCO et ses contraintes globales *GEOST* ou *DIFFN*, reste à faire afin de valider notre approche.

Remerciements

Les auteurs souhaitent remercier leurs partenaires industriels *TBC Générateur d'Innovation*, *Millet* et *SYBois* ainsi que l'ensemble des membres du consortium de projet C.R.I.B.A. pour la construction du premier modèle à base de contraintes supportant cette problématique de calepinage.

Références

[1] Can A. Baykan and Mark S. Fox. Spatial synthesis by disjunctive constraint satisfaction. *Artificial Intelligence for Engineering, Design, Analysis and Manufacturing*, 11 :245–262, 9 1997.

[2] The Energy Conservation Center. *Energy Conservation Handbook*. Japan, 2012.

[3] U.S. Green Building Council. *New Construction Reference Guide*, 2013.

[4] Alwalid N. Elshafei. Hospital layout as a quadratic assignment problem. *Operational Research Quarterly*, 28(1) :pp. 167–179, 1977.

[5] M . Falcon and F . Fontanili. Process modelling of industrialized thermal renovation of apartment buildings. *In eWork and eBusiness in Architecture, Engineering and Construction, European Conference on Product and Process Modelling (ECPPM 2010)*, September 2010.

[6] Y.K. Juan, P. Gao, and J. Wang. A hybrid decision support system for sustainable office building renovation and energy performance improvement. *Energy and Buildings*, 42(3) :290–297, March 2010.

[7] Kyoung Jun Lee, Hyun Woo Kim, Jae Kyu Lee, and Tae Hwan Kim. Case-and constraint-based project planning for apartment construction. *AI Magazine*, 19(1) :pp. 13–24, 1998.

[8] Robin S Liggett. Automated facilities layout : past, present and future. *Automation in Construction*, 9(2) :pp. 197 – 215, 2000.

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

[10] L. Perez-Lombard, J. Ortiz, and C. Pout. A review on buildings energy consumption information. *Energy and Buildings*, 40(3) :394 – 398, 2008.

[11] B. Poel, G. van Cruchten, and C.A. Balaras. Energy performance assessment of existing dwellings. *Energy and Buildings*, 39(4) :393–403, April 2007.

[12] David M. Tate and Alice E. Smith. A genetic approach to the quadratic assignment problem. *Computers and Operations Research*, 22(1) :pp. 73 – 83, 1995.

[13] E. Vareilles, A. F. Barco, M. Falcon, M. Aldanondo, and P. Gaborit. Configuration of high performance apartment buildings renovation : a constraint based approach. In *Conference of Industrial Engineering and Engineering Management (IEEM)*. IEEE., 2013.

[14] M. Zawidzki, K. Tateyama, and I. Nishikawa. The constraints satisfaction problem approach in the design of an architectural functional layout. *Engineering Optimization*, 43(9) :943–966, 2011.

Une comparaison de logiciels d'optimisation sur une large collection de modèles graphiques

D. Allouche[1] **S. de Givry**[1] **B. Hurley**[2] **G. Katsirelos**[1] **B. O'Sullivan**[2] **T. Schiex**[1]

[1] MIAT, UR-875, INRA, F-31320 Castanet Tolosan, France
[2] Insight Centre for Data Analytics, University College Cork, Ireland
simon.degivry@toulouse.inra.fr b.hurley@4c.ucc.ie

Résumé

Le cadre des modèles graphiques à *variables discrètes* permet de modéliser des problèmes d'optimisation NP-difficiles pour lesquels la fonction objectif se factorise en un ensemble de *fonctions locales*. L'interprétation graphique de ces modèles est que chaque fonction est représentée par une clique sur les variables de sa portée. Les modèles graphiques dits *déterministes* ont pour objectif de minimiser la somme des fonctions locales, pouvant aussi être des contraintes si seuls les coûts zéro ou infini sont utilisés. Les modèles graphiques dits *probabilistes* ont pour objectif de maximiser le produit des fonctions (des contraintes si usage uniquement des probabilités zéro ou un). Une transformation directe existe entre les deux classes de modèles graphiques, qui peuvent également être modélisés en programmation linéaire en nombres entiers ou en problème de satisfiabilité maximum en logique propositionnelle.

Dans cet article, nous évaluons plusieurs logiciels implémentant l'état de l'art en méthodes complètes d'optimisation sur une large collection de modèles graphiques déterministes et probabilistes issus de diverses compétitions Max-CSP 2008, Probabilistic Inference Challenge 2011, Weighted Partial Max-SAT Evaluation 2013, MiniZinc Challenge 2012 & 2013, ainsi que des collections provenant des problème de satisfaction de contraintes pondérées et en traitement d'images. Au total, 3018 instances sont mises à disposition dans cinq formats et sept formulations avec les scripts de conversion à `http://genoweb.toulouse.inra.fr/~degivry/evalgm`. Les résultats montrent que différents logiciels généralistes obtiennent de bons résultats sur plusieurs catégories de modèles graphiques, suggérant des opportunités pour une approche portfolio d'algorithmes.

1 Introduction

Les modèles graphiques permettent de représenter une distribution multivariée de manière compacte en exploitant la factorisation de la distribution par des fonctions locales. Nous nous restreignons au cas de variables discrètes. Ce cadre fait habituellement les hypothèses restrictives suivantes : faible arité des fonctions et taille restreinte des domaines permettant d'exprimer des fonctions quelconques en extension.

En intelligence artificielle, le problème de satisfaction de contraintes (CSP) et sa variante pondérée pour l'optimisation (WCSP) consiste à trouver une affectation de toutes les variables qui minimise une distribution définie par la somme de fonctions locales, les contraintes étant capturées par des fonctions à valeurs dans $\{0, \infty\}$. Restreint à des variables Booléennes et le langage de la logique propositionnelle en forme normale conjonctive, le problème de satisfiabilité maximum (Max-SAT) a le même objectif. La programmation par contraintes (PPC) peut également modéliser ces problèmes en introduisant des variables de coût [27].

Les modèles graphiques probabilistes (PGM) [18] appliquent la même idée de factorisation pour représenter une distribution de probabilités d'un ensemble de variables aléatoires. Ils s'expriment à l'aide de deux principaux cadres : celui des réseaux Bayésiens (BN) et celui des champs aléatoires de Markov (MRF). Le problème de trouver une affectation des variables de probabilité maximum, appelé *Maximum Probability Explanation* pour BN ou *Maximum A Posteriori* pour MRF, a de nombreuses applications en particulier en traitement d'images et en bioinformatique. Une simple transformation ($-\log$) convertit ces problèmes en WCSP.

Les modèles graphiques peuvent aussi être modélisés en programmation linéaire à variables 0/1 (01LP), communément utilisée en recherche opérationnelle (RO). Nous considérons deux encodages par la suite. L'un deux se réfère à la notion de *polytope local* [14, 18]. Les algorithmes de propagation de messages, utilisés pour les PGM, peuvent s'interpréter comme des algorithmes de descente de gradient par bloc dans le dual de ce polytope [14]. Ce constat s'applique aussi aux cohérences locales souples dans les modèles graphiques déterministes [8].

Pour les expérimentations, nous avons collecté des instances de modèles graphiques déterministes et probabilistes issues de différentes sources incluant des compétitions CSP, Max-SAT, PPC et PGM. Ces instances ont été encodées dans plusieurs langages de l'IA (WCSP, Max-SAT, MRF), en PPC et en RO (01LP). Habituellement, ces compétitions se restreignent à une famille de méthodes d'optimisation associée à un langage particulier. Au contraire, nous comparons l'efficacité de plusieurs *méthodes complètes* [1] d'optimisation de l'état de l'art pour chacun de ces langages.

2 Langages d'optimisation combinatoire

Dans cette section, nous décrivons les différents langages d'optimisation combinatoire utilisés ensuite. Nous commençons par les modèles graphiques probabilistes qui sont moins connus de la communauté PPC. Nous ne présentons que les champs de Markov car ils n'imposent pas de restriction supplémentaire sur les fonctions locales factorisant la distribution, les réseaux Bayésiens imposant l'utilisation de probabilités conditionnelles avec une exigence de normalisation.

Champ aléatoire de Markov

définition 1 *Un champ aléatoire de Markov (MRF) est défini par une paire (X, Ψ) avec $X = \{x_1, \ldots, x_n\}$, un ensemble de n variables aléatoires et Ψ, un ensemble de* fonctions de potentiel *multiplicatives. Chaque variable $x_i \in X$ a un domaine fini D_i de valeurs qui peuvent lui être affectée. Une fonction de potentiel $\psi_S \in \Psi$, portant sur les variables $S \subseteq X$, est une fonction $\psi_S : D_S \mapsto \mathbb{R}^+$, où D_S exprime le produit Cartésien des domaines D_i pour $x_i \in S$.*

Un champ aléatoire de Markov discret définit une distribution non-normalisée sur X. Pour passer à une distribution de probabilité, on note $P(t)$ la probabilité d'un n-uplet (tuple) donné $t \in D_X$ comme étant :

$$P(t) = \frac{\prod_{\psi_S \in \Psi} \psi_S(t[S])}{Z} = \frac{\exp(-\sum_{\phi_S \in \Phi} \phi_S(t[S]))}{Z}$$

avec $Z = \sum_{t \in D_X} \prod_{\psi_S \in \Psi} \psi_S(t[S])$, une constante de normalisation et $t[S]$, la projection d'un tuple t sur l'ensemble des variables S. La fonction $\phi_S = -\log(\psi_S)$ est appelée fonction de potentiel additive ou aussi *énergie*, en relation avec la physique statistique.

Dans cet article, nous ne considérons que le problème d'optimisation MAP (maximum a posteriori) qui consiste à trouver une affectation complète de probabilité maximum (équivalent à une solution d'énergie minimum). Le problème MAP peut être résolu par des logiciels comme `daoopt` [26], gagnant du PASCAL Probabilistic Inference Challenge (PIC) [2] en 2011, ou `mplp2` [32, 31] qui exploite un algorithme de descente de gradient par bloc pour approcher la résolution d'un système linéaire associé au problème MRF.

exemple 1 *Soit le problème MRF avec deux variables $\{x, y\}$, $D_x = \{a, b\}, D_y = \{a, b, c\}$ et une fonction de potentiel multiplicative $\psi(x, y)$, avec $\psi(a, a) = \psi(b, b) = 1$, $\psi(a, b) = \psi(b, a) = 0.5$, $\psi(a, c) = \psi(b, c) = 0$. L'affectation $(x = a, y = a)$ a une probabilité normalisée maximum $\frac{1}{3}$.*

Problème de satisfaction de contraintes pondérées

Le problème de satisfaction de contraintes pondérées (Weighted CSP) étend le formalisme CSP en remplaçant les contraintes par des fonctions de coût à valeurs entières positives.

définition 2 *Un problème de satisfaction de contraintes pondérées (WCSP) est défini par un triplet (X, W, k) avec $X = \{x_1, \ldots, x_n\}$, un ensemble de n variables discrètes, W, un ensemble de fonctions de coût positives et k, un coût maximum éventuellement infini. Chaque variable $x_i \in X$ a un domaine fini D_i de valeurs qui peuvent lui être affectée. Une fonction $w_S \in W$, portant sur les variables $S \subseteq X$, est une fonction $w_S : D_S \mapsto \{\alpha \in \mathbb{N} \cup \{k\} \ : \ \alpha \leq k\}$.*

Le paramètre k est associé aux tuples interdits permettant de représenter des contraintes. Dans les WCSP, la somme des coûts est bornée par $k : \alpha +_k \beta = \min(\alpha + \beta, k)$. Le coût d'une affectation complète, à minimiser, est égal à la somme bornée de toutes les fonctions de coût. Les WCSP peuvent être résolus par des logiciels comme `toulbar2` [22, 13, 11], gagnant des compétitions Max-CSP [3] en 2008 et MRF en 2010 (UAI 2010 Evaluation [4]).

1. Les méthodes ne sont pas comparées sur la qualité des solutions trouvées mais sur leur capacité à trouver des solutions optimales et prouver leur optimalité dans un temps limité.

2. `http://www.cs.huji.ac.il/project/PASCAL/realBoard.php`

3. `http://www.cril.univ-artois.fr/CPAI08/`

4. `http://www.cs.huji.ac.il/project/UAI10`

Max-SAT Lorsque l'on se restreint à des domaines Booléens avec un langage de clauses pondérées, le problème WCSP devient un problème Max-SAT.

définition 3 *Un problème Max-SAT (Weighted Partial Max-SAT) est défini par un ensemble de paires* $\langle C, w\rangle$*, avec* C*, une clause et* $w \in \mathbb{N} \cup \{\infty\}$*, un nombre appelé le* poids *de la clause. Une clause est une disjonction de littéraux. Un littéral est une variable Booléenne ou sa négation.*

Si le poids d'une clause est infini alors la clause est dite *dure*, sinon elle est *souple*. L'objectif est de trouver une affectation de toutes les variables apparaissant dans les clauses telle que toutes les clauses dures soient satisfaites et que le poids total des clauses souples insatisfaites soit minimum. Parmi les logiciels dédiés à Max-SAT, notons `maxhs` [9, 10], gagnant de la compétition Max-SAT [5] en 2013 en catégorie *crafted WPMS*.

Programmation linéaire en nombres entiers Un problème de programmation linéaire à variables 0/1 (01LP) est défini par un critère linéaire et un système d'équations et inéquations linéaires sur un ensemble de variables Booléennes. Le but est de minimiser le critère tout en satisfaisant l'ensemble des contraintes. Une référence parmi les solveurs 01LP est le logiciel `cplex` d'IBM-ILOG.

Programmation par contraintes Un problème de programmation par contraintes (PPC) est défini par un ensemble de variables discrètes et un ensemble de contraintes. Le but est de minimiser une variable objectif donnée tout en satisfaisant les contraintes. Notons les solveurs PPC `numberjack mistral`, `gecode` et `opturion/cpx`, gagnants respectivement des compétitions CSP 2009 [6], MiniZinc 2012 et MiniZinc 2013 [7].

3 Traduction entre les divers formalismes

Dans cette section, nous présentons l'encodage des différents modèles graphiques issus de l'IA/RO/PPC permettant de passer d'un langage à un autre. Pour les conversions, nous avons appliqué une stratégie en étoile en utilisant le formalisme WCSP comme encodage central.

5. `http://maxsat.ia.udl.cat:81/13/benchmarks/`
6. `http://www.cril.univ-artois.fr/CPAI09`
7. `http://www.minizinc.org/challenge201X/results201X.html` with $X \in \{2, 3\}$.

Champ aléatoire de Markov A partir des définitions d'un problème MRF et WCSP, il est clair que la seule différence porte sur le domaine des fonctions : MRF utilise des fonctions potentielles à valeurs réelles tandis que WCSP est habituellement restreint à l'utilisation d'entiers positifs [8].

La conversion d'une instance WCSP en une instance MRF s'effectue par un passage à l'exponentiel des coûts. La base de l'exponentiel est choisie de manière à avoir le potentiel multiplicatif le plus grand égal à 1. Le coût interdit k correspond à un potentiel multiplicatif à 0, préservant ainsi l'effacement de valeurs dans les domaines des variables. Le résultat de la conversion est une instance MRF dans le format UAI "MARKOV" [9].

A l'inverse, pour passer d'une instance MRF vers une instance WCSP, nous employons une représentation des énergies à virgule fixe (précision de chaque énergie à 2 chiffres après la virgule dans les expérimentations). Un décalage constant sur les valeurs résultantes est effectué pour chaque fonction de coût de manière à n'avoir que des entiers positifs.

exemple 2 *La fonction potentielle de l'exemple 1 se traduit en une fonction de coût* $f(a,a) = f(b,b) = 0$*,* $f(a,b) = f(b,a) = -100\log(0.5) = 30$*,* $f(a,c) = f(b,c) = +\infty$*. L'affectation* $(x = a, y = a)$ *a un coût minimum* 0*.*

Max-SAT Une instance Max-SAT est un cas particulier d'une instance WCSP (les poids des clauses étant aussi des entiers positifs). A l'inverse, nous avons repris deux encodages existants de CSP vers SAT : l'encodage *direct* [3] et l'encodage de *tuple* [4].

Encodage direct : pour chaque variable x_i ayant un domaine de taille $|D_i| > 2$, nous avons une proposition d_{ir} pour chaque valeur $r \in D_i$. Cette proposition est vraie ssi la variable x_i est affectée à la valeur r. Pour cela, nous avons les clauses dures $(\neg d_{ir} \vee \neg d_{is})$, $\forall x_i \in \{x_1, \ldots, x_n\}$, $\forall r < s$, $r, s \in D_i$ (clauses *At Most One*), ainsi qu'une clause dure $(\bigvee_r d_{ir})$, $\forall x_i$ (clauses *At Least One*). Ces clauses imposent qu'exactement une seule valeur soit choisie par variable du WCSP. Les variables Booléennes sont directement encodées par une seule proposition sans clause AMO/ALO. Enfin, nous ajoutons une clause $(\bigvee_{x_i \in S} \neg d_{it[x_i]})$ de poids $w_S(t)$, $\forall w_S \in W$, $\forall t \in D_S$ avec $w_S(t) > 0$.

exemple 3 *L'exemple 1 se traduit par quatre propositions* x, y_a, y_b, y_c*, les clauses AMO/ALO* $(\neg y_a \vee \neg y_b)$*,* $(\neg y_a \vee \neg y_c)$*,* $(\neg y_b \vee \neg y_c)$*,* $(y_a \vee y_b \vee y_c)$ *et les clauses pondérées* $(x \vee \neg y_b, 30), (x \vee \neg y_c, \infty), (\neg x \vee \neg y_a, 30), (\neg x \vee \neg y_c, \infty)$*.*

8. A noter que des rationnels positifs sont aussi utilisés dans [8].
9. `http://graphmod.ics.uci.edu/uai08/FileFormat`

Encodage de tuple : le même encodage des domaines que précédemment est effectué, ainsi que pour les fonctions de coût d'arité nulle ou portant sur une seule variable. Nous introduisons une proposition $p_{S,t}$, $\forall w_S \in W$, $\forall t \in D_S$ uniquement lorsque $|S| > 1$ et $w_S(t) < k$. Si $w_S(t) > 0$, nous ajoutons la clause $(\neg p_{S,t})$ avec le poids $w_S(t)$. Cela représente le coût à payer si le tuple t est utilisé. De plus, nous avons une clause dure $(\neg p_{S,t} \vee d_{it[x_i]})$, $\forall x_i \in S$. Si le tuple t est utilisé, alors les valeurs correspondantes $t[x_i]$ doivent être affectées. Enfin, nous avons une clause dure $(\neg d_{ir} \vee \bigvee_{t \in D_S, t[x_i]=r, w_S(t)<k} p_{S,t})$, $\forall w_S \in W, |S| > 1$, $\forall x_i \in S$, $\forall r \in D_i$. Ces clauses imposent que si une valeur $r \in D_i$ est affectée à x_i, un des tuples $t \in D_S$ avec $t[x_i] = r$ doit être utilisé (ou s'ils sont tous interdits, cette valeur ne peut être affectée). Cet encodage a été proposé initialement pour encoder un CSP en SAT [4]. Il permet que la propagation unitaire sur l'encodage de tuple *supprime* les mêmes valeurs que la cohérence d'arc appliquée au CSP d'origine.

exemple 4 *L'exemple 1 se traduit par huit propositions* $x, y_a, y_b, y_c, p_{x_a y_a}, p_{x_a y_b}, p_{x_b y_a}, p_{x_b y_b}$, *les clauses AMO/ALO* $(\neg y_a \vee \neg y_b)$, $(\neg y_a \vee \neg y_c)$, $(\neg y_b \vee \neg y_c)$, $(y_a \vee y_b \vee y_c)$, *les clauses pondérées* $(\neg p_{x_a y_b}, 30)$, $(\neg p_{x_b y_a}, 30)$ *et les clauses dures* $(\neg p_{x_a y_a} \vee \neg x)$, $(\neg p_{x_a y_a} \vee y_a)$, $(\neg p_{x_a y_b} \vee \neg x)$, $(\neg p_{x_a y_b} \vee y_b)$, $(\neg p_{x_b y_a} \vee x)$, $(\neg p_{x_b y_a} \vee y_a)$, $(\neg p_{x_b y_b} \vee x)$, $(\neg p_{x_b y_b} \vee y_b)$ *et* $(x \vee p_{x_a y_a} \vee p_{x_a y_b})$, $(\neg x \vee p_{x_b y_a} \vee p_{x_b y_b})$, $(\neg y_a \vee p_{x_a y_a} \vee p_{x_b y_a})$, $(\neg y_b \vee p_{x_a y_b} \vee p_{x_b y_b})$, $(\neg y_c)$.

Le résultat de ces deux encodages est donné dans le format `wcnf` [10], qui inclut l'information du majorant k, de manière à préserver la propagation.

La complexité asymptotique des deux encodages est la même en terme de nombre de tuples à représenter explicitement : $O(nd^2 + ert)$ avec n, le nombre de variables, d, la taille du plus grand domaine, e, le nombre de fonctions de coût, r, l'arité maximum des fonctions de coût, et t, le nombre maximum de tuples par fonction, avec l'hypothèse que $t > d$. Cependant cette notation asymptotique cache une constante plus grande pour l'encodage de tuple. Or, le nombre de tuples devant être explicitement représentés dans l'encodage direct est le nombre de tuples ayant un coût non nul, alors que c'est le nombre de tuples ayant un coût inférieur à k dans l'encodage de tuple. Dans nos expérimentations, nous observons que l'encodage de tuple génère des fichiers bien plus volumineux que l'encodage direct, indiquant que pour la plupart des instances, il y a beaucoup plus de tuples de coût zéro que de coût infini (k).

10. `http://www.maxsat.udl.cat/08/index.php?disp=requirements`

Programmation linéaire en nombres entiers L'encodage d'une instance WCSP en 01LP [11] est similaire à celui fait pour Max-SAT. Des variables 0/1 remplacent les variables propositionnelles. Cependant des différences mineures apparaissent dues au pouvoir d'expression supérieur des contraintes linéaires par rapport aux clauses.

Encodage direct : les clauses AMO/ALO sont remplacées par une seule contrainte linéaire : $\sum_{r \in D_i} d_{ir} = 1, \forall x_i \in X$ avec $|D_i| > 2$. La fonction objectif d'un WCSP n'étant pas forcément linéaire, cela nécessite d'introduire des variables 0/1 $p_{S,t}, \forall w_S \in W$, $\forall t \in D_S$ avec $0 < w_S(t) < k$. La fonction objectif de la formulation 01LP est alors $\sum w_S(t) p_{S,t}$. Nous ajoutons une contrainte linéaire $\sum_{x_i \in S}(1 - d_{it[x_i]}) + p_{S,t} \geq 1$ qui impose que la variable $p_{S,t} = 1$ si le tuple t est utilisé. Si $w_S(t) = k$, alors la contrainte devient $\sum_{x_i \in S}(1 - d_{it[x_i]}) \geq 1$ et le terme $w_S(t) p_{S,t}$ correspondant disparaît de l'objectif.

exemple 5 *L'exemple 1 se traduit par six variables 0/1* $x, y_a, y_b, y_c, p_{x_a y_b}, p_{x_b y_a}$, *les contraintes* $y_a + y_b + y_c = 1$, $x - y_b + p_{x_a y_b} \geq 0$, $-x - y_a + p_{x_b y_a} \geq -1$, $x - y_c \geq 0$, $-x - y_c \geq -1$ *et la fonction objectif* $30 p_{x_a y_b} + 30 p_{x_b y_a}$.

Encodage de tuple : nous utilisons le même encodage que précédemment pour exprimer les domaines et les fonctions de coût d'arité 0/1. Nous ajoutons une contraint linéaire $d_{ir} = \sum_{t \in D_S, t[x_i]=r, w_S(t)<k} p_{S,t}$, $\forall w_S \in W, |S| > 1$, $\forall x_i \in S, \forall r \in D_i$ qui impose que $d_{ir} = 1$ ssi il existe un tuple t tel que $t[x_i] = r$ et $w_s(t) < k$ (et $d_{ir} = 0$ si tous les tuples sont interdits). La fonction objectif est identique à l'encodage direct.

exemple 6 *L'exemple 1 se traduit par huit variables 0/1* $x, y_a, y_b, y_c, p_{x_a y_a}, p_{x_a y_b}, p_{x_b y_a}, p_{x_b y_b}$, *les contraintes* $y_a + y_b + y_c = 1$, $1 - x = p_{x_a y_a} + p_{x_a y_b}$, $x = p_{x_b y_a} + p_{x_b y_b}$, $y_a = p_{x_a y_a} + p_{x_b y_a}$, $y_b = p_{x_a y_b} + p_{x_b y_b}$, $y_c = 0$ *et la fonction objectif* $30 p_{x_a y_b} + 30 p_{x_b y_a}$.

Cet encodage de tuple a été proposé par Koster [20]. Il est équivalent à l'encodage en 01LP couramment utilisé pour les PGM dans le cas de fonctions d'arité au plus 2 [18] (chapitre 13 section 5). Il est facile de voir que la contrainte d'intégralité sur les variables $p_{S,t}$ peut être relâchée dans les deux encodages : si toutes les variables d_{ir} sont affectées à 0 ou 1, alors les contraintes et la fonction objectif assurent que les $p_{S,t}$ sont fixées à 0 ou 1. Dans le format `cplex` "LP" généré, nous relaxons cette contrainte d'intégralité.

Il est intéressant de noter que la relaxation continue de l'encodage de tuple est équivalente au *polytope local* décrit pour les MRFs [33, 14] et précédemment

11. Nous n'étudions pas l'encodage inverse 01LP vers WCSP.

étudié par Schlesinger en traitement d'images pour le cas de grammaires de motifs 2D [30]. C'est aussi équivalent au dual du programme linéaire lié à la cohérence d'arc optimale souple (OSAC) [7, 8]. Le minorant produit par OSAC est supérieur à ceux produits par les autres cohérences d'arc souples comme EDAC [22], à l'exception des possibles interactions avec la cohérence de noeud. Ce minorant dual a récemment été montré comme étant très expressif au sens où n'importe quel programme linéaire peut être traduit en temps linéaire dans le problème de calculer ce minorant pour un modèle graphique adapté [19].

Programmation par contraintes Dans [27], un encodage d'une instance WCSP en PPC a été proposée. Les variables de décision sont identiques au WCSP. Chaque fonction de coût est réifiée en une contrainte qui s'applique aux variables de la fonction et à une variable supplémentaire représentant le coût de l'affectation. Le problème résultant est un CSP avec plus de variables et des arités accrues. Typiquement, les fonctions de coût d'arité 1 et 2 sont traduites en des contraintes de `table` d'arité 2 et 3 respectivement. Enfin une variable objectif supplémentaire est reliée par une contrainte de `somme` à toutes les autres variables de coût. Toutes ces variables supplémentaires ont un domaine de valeurs entières positives borné par le majorant initial k. La même approche est employée pour encoder une instance Max-SAT, à l'exception des contraintes de `table` qui sont remplacées par des expressions Booléennes réifiées encodant les clauses dures et souples. Le résultat est exprimé dans le langage PPC `minizinc` [25].

A l'inverse, traduire une instance PPC en WCSP est une tâche plus difficile. Il s'agit de retrouver la factorisation de la fonction objectif en une somme de fonctions de coût locales, à partir de la variable objectif, tout en éliminant les variables intermédiaires. Pour cela, nous avons développé un prototype dans `numberjack` [15] lisant le format bas-niveau `flatzinc` [12] [25]. De plus, les contraintes globales ont été décomposées en fonctions de coût ($\{0, \infty\}$) d'arité 3 [1].

4 Comparaison de logiciels d'optimisation

4.1 Collection de benchmarks

Nous avons collecté 3018 instances provenant de différentes sources de modèles graphiques déterministes (WCSP, CSP, Max-SAT), probabilistes (MRF) et aussi de compétitions PPC. Elles sont disponibles aux formats `uai`, `wcsp`, `wcnf`, `lp` et `minizinc` [13].

CFLib [14] est une collection de problèmes WCSP. Nous en avons extrait une partie, codée nativement au format `wcsp` et recodée manuellement au format `minizinc`. Cela inclut les enchères combinatoires [23], l'allocation de fréquence CELAR/GRAPH [6], la correction d'erreur Mendélienne [29], la conception de protéines [2], la sélection de prises de vue du satellite SPOT5 [5] et l'allocation d'entrepôts [21, 22].

Les instances probabilistes MRF proviennent des compétitions UAI 2008 (problème d'analyse de liaison génétique [15] [13], arité 5) et PIC 2011 (arité 2), au format natif `uai` (converties à 2 chiffres après la virgule). D'autres instances d'arité 2 et 3 sont issues d'un benchmark OpenGM2 Computer Vision and Pattern Recognition (CVPR) [16] [16] au format natif `HDF5` contenant des énergies au lieu des probabilités. ColorSeg, MatchingStereo, PhotoMontage ont des énergies entières directement traduites en fonctions de coût, les autres problèmes étant convertis à 8 chiffres après la virgule (instances limitées à 1Go).

Les instances Max-SAT proviennent des catégories Crafted (W)PMS (MIPLib & DIMACS Max Clique) et Industrial WPMS [17]. La conversion au format `wcsp` permet d'encoder chaque clause par une fonction de coût avec un seul tuple de coût non nul. Les traductions vers la PPC (resp. MRF) imposent que les coûts tiennent sur 32 bits (resp. les fonctions de coût soient de faible arité).

Nous avons sélectionné des problèmes CSP binaires définis en extension (*i.e.,* sans contrainte globale) et comportant des instances insatisfiables (BlackHole, Langford, Quasi-group Completion Problem, coloriage de graphe et problèmes aléatoires Composed, 3-SAT EHI et Geometric) des compétitions CSP & Max-CSP [18]. Ces instances au format natif `xcsp2.1/xml` ont été transformées en WCSP (Max-CSP) tels que chaque tuple permis (resp. interdit) d'une contrainte a un coût nul (resp. unitaire) dans la fonction de coût correspondante. Nous fixons $k = 1000$. Enfin, nous avons pris des problèmes PPC décomposables en WCSP des compétitions MiniZinc 2012 & 2013.

Dans la Table 1, nous indiquons les tailles maximum des instances par problème. Dans le cas des problèmes issus de la PPC, ces tailles correspondent au sous-ensemble des instances décomposables (seule une partie indiquée entre parenthèses des instances FastFood, Golomb et OnCallRostering a pu être convertie au format `wcsp` en utilisant moins de 1 Go par instance).

12. Dans les expérimentations, une limite de 20 minutes a été imposée lors de la traduction de `minizinc` à `flatzinc`.
13. `http://genoweb.toulouse.inra.fr/~degivry/evalgm`
14. `http://costfunction.org/benchmark`
15. `http://graphmod.ics.uci.edu/uai08/Evaluation/Report/Benchmarks`
16. `http://hci.iwr.uni-heidelberg.de/opengm2`
17. `http://maxsat.ia.udl.cat:81/13/benchmarks/`
18. `http://www.cril.univ-artois.fr/CPAI08/, ../~lecoutre/benchmarks.html`

Nous donnons également les majorants initiaux maximum k, arbitrairement fixés à une grande valeur ou à la somme du maximum de chaque fonction de coût plus un, ainsi qu'une sur-approximation de l'ensemble des coûts utilisés dans un problème avec ∞ indiquant l'occurrence de tuples interdits (contraintes).

Les plus grandes instances en nombre de variables proviennent des benchmarks Max-SAT et CVPR (presque 1 million de variables pour Max-SAT/TimeTabling et un demi-million pour CVPR/PhotoMontage et ColorSeg). De plus, Max-SAT contient des clauses portant sur un grand nombre de variables (580 dans Haplotyping). Pour les autres benchmarks, l'arité des fonctions reste faible entre 2 et 5. La connectivité du graphe des modèles graphiques probabilistes (MRF/CVPR) et Max-SAT est souvent très faible (utilisation d'une structure de grille pour CVPR). Cependant MRF/ObjectDetection, WCSP/ProteinDesign, Max-CSP/Langford et CVPR/Matching ont un graphe complet. MRF/ProteinFolding a le plus grand domaine (503 valeurs). La plupart des instances CVPR utilisent des coûts dans un intervalle très large (du fait de la précision à 8 chiffres), tandis que les instances Max-CSP ne contiennent que des coûts 0/1. Tous les modèles graphiques déterministes, à l'exception de Max-CSP et WCSP/CELAR, ont des tuples interdits. A l'inverse, les modèles probabilistes MRF & CVPR n'en ont pas (exceptés MRF/Linkage & DBN).

4.2 Logiciels et paramètres expérimentaux

Nous avons comparé plusieurs logiciels de l'état de l'art en optimisation combinatoire : `daoopt` [19] (réglage des paramètres à 20 min. sauf pour CVPR à 1 heure [26], sans amélioration des bornes par MPLP), `toulbar2` [20] (paramètres `-l=1 -dee=1`), `mplp2` [21] (seuil de précision interne à 2.10^{-7}), `maxhs` (pas de paramètre), `numberjack mistral` [22], `gecode` [23], `opturion/cpx` [24] (mode *free search*) et `cplex` 12.4 (paramètres EPAGAP, EPGAP, and EPINT mis à zéro pour éviter un arrêt intempestif).

Tous les calculs ont été réalisés sur un seul coeur AMD Operon 6176 à 2.3 GHz et 8 Go de mémoire.

4.3 Résultats expérimentaux

En Table 2 figure le nombre d'instances résolues en moins de 20 min. [25] (excepté 1 heure pour CVPR) [26].

Bien que le meilleur solveur en nombre d'instances résolues par classe de problèmes appartienne à cette classe, *i.e.,* `toulbar2` pour WCSP, `maxhs` pour Max-SAT, `gecode` pour PPC et `mplp2` pour CVPR, les résultats montrent que certains solveurs comme `maxhs`, `toulbar2` et `cplex` marchent bien sur plusieurs classes, résolvant resp. 1942, 2191 et 2323 instances parmi 3018, gagnant la première place pour 12, 17, 18 problèmes respectivement [27] parmi 46 catégories de problèmes. CVPR/ColorSeg est le problème avec l'espace de recherche le plus grand ($d^n = 2^{829440}$) complètement résolu par `mplp2`. MRF/ObjectDetection est le problème avec l'espace de recherche le plus petit entièrement non résolu ($d^n \approx 2^{263}$).

`toulbar2` a obtenu la première place pour quatre problèmes MRF (hors CVPR). L'analyse des temps de calcul (figures non disponibles) sur le problème Segmentation montre que `toulbar2` est souvent le plus rapide mais il est dominé par `mplp2` et $\texttt{cplex}_t$ (avec l'encodage de tuple) sur les instances les plus difficiles. Ici, `cplex` avec l'encodage direct et `maxhs` ne résolvent aucune instances avec une taille de domaine $d = 21$ et `daoopt` seulement 7. Sur Linkage [17], `cplex`, suivi par `maxhs`, obtient de très bons résultats, montrant sa capacité à traiter des fonctions de coût d'arité supérieure (5). A noter qu'un solveur Max-SAT, `maxhs` [9, 10], utilisant des techniques peu connues de la communauté MRF, obtient de bons résultats sur les modèles graphiques probabilistes (au moins pour une précision à 2 chiffres). De manière surprenante, l'encodage direct donne de meilleurs résultats sur le problème Grid ($d = 2$) pour `cplex` qui exploite un grand nombre de coupes *zero-half*. Les solveurs PPC obtiennent de mauvais résultats sur MRF en partie du fait de l'absence de contraintes dures (excepté pour Linkage) et de la faiblesse des minorants obtenus par propagation sur les variables de coût, ce phénomène étant aussi observé pour le cas des fonctions de coût globales dans [24].

Sur le benchmark Max-SAT, les clauses d'arité large interdisent d'appliquer l'encodage de tuple (trop grand nombre de tuples de coût nul) ni d'exprimer les problèmes en utilisant des tables en extension comme pour le format `uai`. Il est intéressant de noter la complémentarité de `cplex` et `maxhs` (qui utilise aussi `cplex` pour extraire un minorant à partir de noyaux insatisfiables identifiés par `minisat`) suivant les problèmes. Sur le problème Upgradeability, `cplex` (resp. `toulbar2`) est jusqu'à deux (resp. un) ordre(s) de grandeur plus rapide que `maxhs`. Dans PlanningWPref, `opturion/cpx`, un solveur PPC intégrant un mécanisme d'apprentissage à partir des conflits, obtient la seconde place derrière `maxhs`, résolvant toutes les instances (ex-

19. `https://github.com/lotten/daoopt` version 1.1.2.
20. `mulcyber.toulouse.inra.fr/projects/toulbar2` v0.9.6
21. `http://cs.nyu.edu/~dsontag/` version 2.
22. `http://numberjack.ucc.ie/` version 1.3.40.
23. `http://www.gecode.org/` version 4.2.0.
24. `http://www.opturion.com` version 1.0.2.
25. Sans les temps de conversion entre formats.
26. Des résultats détaillés par instance et des figures *cactus plot* à `http://genoweb.toulouse.inra.fr/~degivry/evalgm`.
27. En prenant le meilleur encodage à chaque fois pour `maxhs` et `cplex`.

cepté WCNF_storage_p03) qui ont pu être traduites dans le format bas-niveau `flatzinc` en moins de 20 minutes, incluant une instance avec 10946 variables (WCNF_pathways_p18).

Concernant le benchmark WCSP, `toulbar2` domine clairement sur les problèmes CELAR, Pedigree et ProteinDesign, tandis que `cplex` avec l'encodage direct, suivi par `maxhs`, l'emporte sur les problèmes Auction et Warehouse issus de la Recherche Opérationnelle. L'encodage de tuple en 01LP se comporte favorablement si la taille des instances est relativement faible ($n \times d \leq 20,000$), sinon des problèmes d'allocation mémoire apparaissent comme c'est le cas pour les plus grandes instances Warehouse.

Les bonnes performances de `maxhs` dans le benchmark Max-CSP peuvent s'expliquer par sa capacité à bien résoudre toutes les instances satisfiables (optimum à zéro) en particulier pour les problèmes Geometric et QCP grâce à son solveur SAT interne `minisat`. Les bons résultats obtenus par `daoopt` peuvent également s'expliquer par sa phase initiale de recherche locale stochastique [26], trouvant de bons majorants pour la phase suivante de recherche arborescente, spécialement sur les instances aléatoires comme Geometric et EHI. `toulbar2` et `cplex` gagnent la première place sur quatre problèmes Max-CSP, obtenant des performances comparables sur Coloring. Pour le problème Composed, `toulbar2` domine souvent en temps de plusieurs ordres de grandeur par rapport aux autres approches à l'exception du solveur PPC `mistral`. Tous les deux utilisent une heuristique de choix de variable *dom/wdeg* exploitant les conflits particulièrement efficace pour ce problème. Enfin, l'encodage de tuple s'avère toujours contre-productif pour ces problèmes.

Les instances PPC sont pour la plupart difficiles à traduire en fonctions de coût locales et avec de petits domaines ($\frac{1}{4}$ des instances n'ont pu être converties à cause de problèmes d'espace mémoire). De plus, le résultat de la conversion n'est souvent pas approprié pour les solveurs 01LP (les contraintes linéaires étant décomposées), ce qui explique les mauvais résultats pour `cplex`. Sur ce benchmark, `gecode` obtient en moyenne les meilleurs résultats. Cependant, un solveur Max-SAT, `maxhs`, le domine sur deux problèmes : Amaze et OnCallRostering. Et `daoopt` est plus rapide que `gecode` sur les instances difficiles de ParityLearning. En effet, `daoopt` résout toutes les instances en prétraitement grâce à l'élimination de variables [12], nécessitant ici un espace mémoire (2Go) juste inférieur à sa limite fixée par sa borne $i = 25$ [26].

Pour comparer les résultats sur le benchmark MRF/CVPR, au lieu de donner les résultats des solveurs PPC (probablement nuls à cause de problèmes de conversion des coûts en simples domaines bornés à 32 bits), nous indiquons les meilleurs résultats obtenus lors d'une évaluation de méthodes développées en traitement d'images [16] [28]. Les résultats montrent une relative bonne performance des solveurs généralistes tels que `mplp2`, `toulbar2` et `cplex` comparés à des solveurs spécialisés en traitement d'images. Sur le problème Scene Decomposition, qui utilise un modèle de *superpixel* [16], `toulbar2` résout toutes les 715 instances en $0,023$ secondes en moyenne comparé à $0,134$ (resp. $1,039$) secondes pour `mplp2` (resp. `cplex`$_t$). Bien que l'encodage de tuple fut toujours contre-productif pour un solveur Max-SAT sur les autres benchmarks (voir la colonne `maxhs`$_t$ dans la Table 2), ce n'est pas le cas sur ce problème (et aussi pour GeomSurf-3). La plus grande instance résolue par `toulbar2` est InPainting-4/triplepoint4-plain-ring avec 14400 variables en $1,47$ secondes comparé à $157,22$ (resp. $193,72$) secondes pour `mplp2` (resp. `cplex`$_t$). Ici les algorithmes spécialisés dominent avec $0,58$ sec. pour la méthode BUNDLE-H, $2,19$ sec. pour TRWS et $12,39$ sec. pour MCA.

5 Conclusion, vers une approche portfolio

En résumé, `toulbar2` obtient les meilleurs résultats lorsque les domaines sont larges ($d > 20$, excepté BlackHole), `maxhs` lorsque l'arité des clauses est grande ($r > 300$), `daoopt` lorsque la structure du modèle graphique est bien décomposable (faible largeur induite), `mplp2` lorsque la relaxation linéaire est proche de l'optimum, `gecode` lorsque le problème est directement modélisé en contraintes. Pour `cplex`, l'encodage de tuple domine l'encodage direct, sauf pour des domaines de taille 2 (ou 3 pour SPOT5) ou des problèmes de taille trop importante ($n \times d \geq 20,000$). C'est l'inverse pour `maxhs`, sauf en traitement d'images.

Les résultats montrant que le meilleur solveur varie beaucoup en fonction de chaque problème, cela suggère d'élaborer une stratégie portfolio. Nous avons utilisé pour cela Numberjack [15] qui offre une interface vers des solveurs PPC, WCSP, LP et SAT. Nous avons intégré la lecture du format `flatzinc` et combiné deux solveurs (`mistral` avec trois politiques de restart différentes et `toulbar2`), exécutés chacun isolément (1 coeur et 8Go) en parallèle, à l'instar de `ppfolio` [28]. Notre approche résout 644 instances hormis MRF/CVPR (`nj-port` en Table 2). Davantage de travail reste à faire pour mieux parser le format `flatzinc` et ajouter d'autres solveurs comme `cplex` avec les deux encodages direct et de tuple.

28. Dans leur expérimentations, ils ont utilisé un processeur Xeon W3550 à 3GHz, 12Go de mémoire vive et 1 heure de temps de calcul.

Problème	Nb.	n	d	e	r	intervalle des coûts utilisés (k)
MRF (`uai`)	319					
Linkage	22	1289	7	2184	5	18 ... ∞ (1000000)
DBN	108	1094	2	22793	2	7 ... ∞ (100000000)
Grid	21	6400	2	19200	2	27 ... 2967 (100000000)
ImageAlignment	10	400	93	3563	2	907 ... 2291 (100000000)
ObjectDetection	37	60	21	1830	2	530 ... 27860 (100000000)
ProteinFolding	21	1972	503	8816	2	53 ... 23327 (100000000)
Segmentation	100	237	2/21	886	2	1 ... 1166 (100000000)
Max-SAT (`wcnf`)	427					
MIPLib	12	24776	2	107956	93	1 ... ∞ (547769)
MaxClique	62	3321	2	378247	2	1 ... ∞ (3321)
Haplotyping	100	216117	2	1188220	580	1 ... ∞ (10691000)
PackupWeighted	99	25554	2	70677	177	60 ... ∞ (27206200)
PlanningWithPref	29	69409	2	771883	372	3000 ... ∞ (100000)
TimeTabling	25	903884	2	2912880	36	2 ... ∞ (14850)
Upgradeability	100	18169	2	105097	77	1 ... ∞ (61594000000)
WCSP (`wcsp`)	281					
Auction	170	246	2	11528	2	36 ... ∞ (249794)
CELAR	16	458	44	2335	2	1 ... 2020000 (468527000)
Pedigree	10	10017	28	18875	3	1 ... ∞ (2484)
ProteinDesign	10	18	198	171	2	2304 ... ∞ (28925100)
SPOT5	20	1057	4	21786	3	2 ... ∞ (635869)
Warehouse	55	1100	300	101100	2	5124 ... ∞ (238093000)
Max-CSP (`xcsp`)	503					
BlackHole	37	205	50	1651	2	1 (1000)
Coloring	22	450	6	6164	2	1 (1000)
Composed	80	83	10	785	2	1 (1000)
EHI	200	315	7	4715	2	1 (1000)
Geometric	100	50	20	605	2	1 (1000)
Langford	4	33	29	517	2	1 (1000)
QCP	60	264	9	2662	2	1 (1000)
PPC (`minizinc`)	35					
AMaze	6	1573	17	3173	4	1 ... ∞ (100000000)
FastFood	6(1)	2	5	3	2	1 ... ∞ (100000000)
Golomb	6(3)	44	163	717	3	1 ... ∞ (100000000)
OnCallRostering	5(3)	2205	89	4513	4	1 ... ∞ (100000000)
ParityLearning	7	759	20	1440	4	1 ... ∞ (100000000)
VehicleRoutingProb.	5	11531	100	22999	4	12 ... ∞ (100000000)
MRF/CVPR (`hdf5`)	1453					
ChineseChars	100	17856	2	553726	2	17432999 ... 723775999 (210467017114895)
ColorSeg	3	414720	4	2069714	2	5 ... 1280 (632538770)
ColorSeg-4	9	86400	12	258600	2	3717 ... 300000000 (25526293828226)
ColorSeg-8	9	86400	12	430202	2	3717 ... 300000000 (24010445405740)
GeomSurf-3	300	1133	3	5039	3	33175 ... 1517620002 (1082559978572)
GeomSurf-7	300	1133	7	5039	3	406409 ... 1517620002 (1703380258076)
InPainting-4	2	14400	4	42960	2	78539816 ... 10000000000 (118363097154663)
InPainting-8	2	14400	4	71282	2	27768018 ... 10000000000 (118027994393422)
Matching	4	20	20	210	2	3 ... 1000000000000000 (190000000000000000)
MatchingStereo	2	166222	20	497849	2	1 ... 765 (149114978)
ObjectSeg	5	68160	8	203947	2	99 ... 620000000 (77323343626999)
PhotoMontage	2	514080	7	1540689	2	1 ... 100000 (152615400000)
SceneDecomp	715	208	8	769	2	187000 ... 3529616658 (1225793288120)

TABLE 1 – Tailles maximum par problème (n, nombre de variables, d, taille des domaines, e, nombre de fonctions de coût, r, arité des fonctions de coût) et intervalles des coûts avec le majorant initial entre parenthèses (k).

Problème	Nb.	daoopt	mplp2	toulbar2	cplex	cplex$_t$	maxhs	maxhs$_t$	opturion	gecode	mistral	nj-port
MRF (`uai`)	319	144	123	**219**	152	205	104	72	1	0	1	181
Linkage	22	16	1	13	14	**22**	20	20	0	0	1	10
DBN	108	60	0	**77**	64	65	30	0	0	0	0	70
Grid	21	2	2	0	**15**	0	4	2	0	0	0	0
ImageAlignment	10	9	**10**	**10**	0	9	0	0	0	0	0	5
ObjectDetection	37	0	0	0	0	0	0	0	0	0	0	0
ProteinFolding	21	0	10	**19**	9	9	0	0	0	0	0	5
Segmentation	100	57	**100**	**100**	50	**100**	50	50	1	0	0	91
Max-SAT (`wcnf`)	427	11	0	195	269	N/A	**277**	N/A	27	19	15	45
MIPLib	12	2	0	**3**	**3**	N/A	**3**	N/A	2	**3**	2	**3**
MaxClique	62	9	0	33	**38**	N/A	36	N/A	10	15	13	24
Haplotyping	100	N/A	N/A	1	18	N/A	**25**	N/A	MZN	MZN	MZN	MZN
PackupWeighted	99	N/A	N/A	52	**99**	N/A	85	N/A	1	0	0	18
PlanningWPref	29	N/A	N/A	6	11	N/A	**27**	N/A	14	1	0	0
TimeTabling	25	N/A	N/A	0	0	N/A	**1**	N/A	MZN	MZN	MZN	MZN
Upgradeability	100	N/A	N/A	**100**	**100**	N/A	**100**	N/A	N/A	N/A	N/A	N/A
WCSP (`wcsp`)	281	188	44	**247**	241	235	227	195	70	130	111	179
Auction	170	158	0	167	**170**	**170**	**170**	166	61	104	108	**170**
CELAR	16	3	0	**12**	0	3	0	0	1	0	1	1
Pedigree	10	4	0	**10**	5	9	5	5	4	0	0	0
ProteinDesign	10	4	7	**9**	0	6	0	0	0	0	0	5
SPOT5	20	6	0	4	**16**	10	5	5	1	0	2	3
Warehouse	55	13	37	45	**50**	37	47	19	3	26	0	0
Max-CSP (`xcsp`)	503	173	0	216	199	50	**238**	150	118	6	96	221
BlackHole	37	10	0	10	**30**	10	10	10	10	0	10	**30**
Coloring	22	16	0	**17**	**17**	15	12	12	8	4	6	15
Composed	80	26	0	**80**	**80**	15	**80**	15	**80**	0	**80**	**80**
EHI	200	0	0	0	0	0	0	0	0	0	0	0
Geometric	100	91	0	**93**	49	0	88	77	9	0	0	82
Langford	4	**2**	0	**2**	**2**	1	**2**	**2**	1	**2**	0	0
QCP	60	28	0	14	21	9	**46**	34	10	0	0	14
PPC (`minizinc`)	35	10	1	13	2	5	16	8	18	**26**	18	18
AMaze	6	0	0	2	0	2	**6**	3	5	4	2	2
FastFood	6	1	1	1	1	1	1	1	**6**	**6**	5	5
Golomb	6	0	0	3	0	0	3	1	4	**6**	5	5
OnCallRoster.	5	2	0	2	1	2	**3**	**3**	2	2	2	2
ParityLearning	7	**7**	0	5	0	0	3	0	1	**7**	4	4
VRP	5	0	0	0	0	0	0	0	0	**1**	0	0
CVPR (`hdf5`)	1453	1272	**1339**	1301	372	1329	311	1008	`TRWS`	`BUN-DLE`	`MCA`	`MCBC`
ChineseChars	100	0	0	0	0	0	0	0	0	N/A	N/A	**56**
ColorSeg	3	1	**3**	0	0	1	0	0	1	0	**3**	N/A
ColorSeg-4	9	0	**8**	0	0	3	0	0	7	7	**8**	N/A
ColorSeg-8	9	0	4	0	0	2	0	0	2	1	**8**	N/A
GeomSurf-3	300	**300**	**300**	**300**	**300**	**300**	236	291	N/A	277	N/A	N/A
GeomSurf-7	300	252	**300**	281	72	**300**	74	2	N/A	180	N/A	N/A
InPainting-4	2	0	**1**	**1**	0	**1**	0	0	**1**	**1**	**1**	N/A
InPainting-8	2	0	**1**	0	0	0	0	0	0	0	**1**	N/A
Matching	4	**4**	**4**	**4**	0	3	0	0	0	0	N/A	N/A
MatchingStereo	2	0	0	0	0	0	0	0	0	**1**	N/A	N/A
ObjectSeg	5	0	3	0	0	4	0	0	**5**	3	**5**	N/A
PhotoMontage	2	0	0	0	0	0	0	0	0	0	N/A	N/A
SceneDecomp	715	**715**	**715**	**715**	0	**715**	1	**715**	712	673	N/A	N/A
Nb. of 1st Pos.	46	5	10	**17**	13	7	11	3	2	6	1	4

TABLE 2 – Nombre d'instances résolues en moins de 20 min. (1 heure pour CVPR). Meilleurs résultats en gras.

Remerciements.

Ce travail a été en partie financé par l'Agence nationale de la Recherche, référence ANR-10-BLA-0214. Nous sommes reconnaissant à la plateforme bioinformatique Genotoul de Toulouse pour l'accès au cluster.

Références

[1] D Allouche, C Bessiere, P Boizumault, S Givry, P Gutierrez, S Loudni, JP Métivier, and T Schiex. Decomposing global cost functions. In *Proc. of AAAI*, 2012.

[2] D Allouche, S Traoré, I André, S Givry, G Katsirelos, S Barbe, and T Schiex. Computational protein design as a cost function network optimization problem. In *Proc. of CP*, pages 840–849, 2012.

[3] J Argelich, A Cabiscol, I Lynce, and F Manyà. Encoding Max-CSP into partial Max-SAT. In *Proc. of ISMVL*, pages 106–111, 2008.

[4] F Bacchus. GAC via unit propagation. In *Proc. of CP*, pages 133–147, 2007.

[5] E Bensana, M Lemaître, and G Verfaillie. Earth observation satellite management. *Constraints*, 4(3) :293–299, 1999.

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

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

[8] M Cooper, S Givry, M Sanchez, T Schiex, M Zytnicki, and T Werner. Soft arc consistency revisited. *Artif. Intell.*, 174 :449–478, 2010.

[9] J Davies and F Bacchus. Solving MAXSAT by solving a sequence of simpler SAT instances. In *Proc. of CP*, pages 225–239, 2011.

[10] J Davies and F Bacchus. Exploiting the power of MIP solvers in MaxSAT. In *Proc. of SAT*, pages 166–181, 2013.

[11] S de Givry, S Prestwich, and B O'Sullivan. Dead-end elimination for weighted CSP. In *Proc. of CP*, pages 263–272, 2013.

[12] R Dechter. Bucket elimination : A unifying framework for reasoning. *Artificial Intelligence*, 113(1–2) :41–85, 1999.

[13] A Favier, S Givry, A Legarra, and T Schiex. Pairwise decomposition for combinatorial optim. in graphical models. In *Proc. of IJCAI*, 2011.

[14] A Globerson and T Jaakkola. Fixing max-product : Convergent message passing algorithms for MAP LP-relaxations. In *Proc. of NIPS*, pages 553–560, 2007.

[15] E Hebrard, E O'Mahony, and B O'Sullivan. Constraint Programming and Combinatorial Optimisation in Numberjack. In *Proc. of CP-AI-OR*, pages 181–185, 2010.

[16] J Kappes, B Andres, F Hamprecht, C Schnörr, S Nowozin, D Batra, S Kim, B Kausler, J Lellmann, N Komodakis, and C Rother. A comparative study of modern inference techniques for discrete energy minimization problem. In *Proc. of CVPR*, 2013.

[17] A Kishimoto and R Marinescu. Recursive best-first and/or search with overestimation for genetic linkage analysis. In *Proc. of CP Workshop on Constraint Based Methods for Bioinformatics*, 2013.

[18] D Koller and N Friedman. *Probabilistic graphical models : principles and techniques*. The MIT Press, 2009.

[19] V Kolmogorov. The power of linear programming for finite-valued csps : a constructive characterization. In *Automata, Lang., and Program.*, pages 625–636. 2013.

[20] A Koster. *Frequency assignment : Models and Algorithms*. PhD thesis, 1999.

[21] J Kratica, D Tošic, V Filipović, and I Ljubić. Solving the simple plant location problem by genetic alg. *RAIRO*, 35(1) :127–142, 2001.

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

[23] J Larrosa, F Heras, and S de Givry. A logical approach to efficient max-sat solving. *Artif. Intell.*, 172(2-3) :204–233, 2008.

[24] J Lee and K Leung. Consistency techniques for flow-based projection-safe global cost functions in weighted constraint satisfaction. *JAIR*, 43 :257–292, 2012.

[25] N Nethercote, P Stuckey, R Becket, S Brand, G Duck, and G Tack. MiniZinc : Towards a standard CP modelling language. In *Proc. of CP*, pages 529–543, 2007.

[26] L Otten, A Ihler, K Kask, and R Dechter. Winning the PASCAL 2011 MAP challenge with enhanced AND/OR branch-and-bound. In *NIPS DISCML Workshop*, 2012.

[27] T Petit, JC Régin, and C Bessière. Meta constraints on violations for over constrained problems. In *Proc. of ICTAI*, pages 358–365, 2000.

[28] O Roussel. Description of ppfolio. *SAT Competition 2011*, 2011.

[29] M Sánchez, S de Givry, and T Schiex. Mendelian error detection in complex pedigrees using weighted constraint satisfaction techniques. *Constraints*, 13(1-2) :130–154, 2008.

[30] M.I. Schlesinger. Syntactic analysis of two-dimensional visual signals in noisy conditions. *Kibernetika*, 4 :113–130, 1976.

[31] D Sontag, D Choe, and Y Li. Efficiently searching for frustrated cycles in MAP inference. In *Proc. of UAI*, pages 795–804, 2012.

[32] D Sontag, T Meltzer, A Globerson, Y Weiss, and T Jaakkola. Tightening LP relaxations for MAP using message-passing. In *Proc. of UAI*, 2008.

[33] T Werner. A linear programming approach to max-sum problem. *Pattern A. & Mach. Int.*, 29(7), 2007.

CSP Object Model : un assistant de modélisation indépendant des solveurs

Julien Vion

Université de Valenciennes et du Hainaut Cambrésis, LAMIH CNRS UMR 8201
julien.vion@univ-valenciennes.fr

Résumé

CSPOM est un modèle objet permettant de représenter un problème de satisfaction de contraintes. L'objectif de CSPOM est de constituer une interface unifiée entre un grand nombre de solveurs et de langages de modélisation. Pour maximiser l'indépendance de CSPOM vis-à-vis de tout solveur, le modèle objet est minimal : aucune hypothèse n'est faite sur les contraintes supportées par les solveurs. Une seule classe généralise toutes les contraintes, et quelques types de variables standard facilitent le travail de modélisation.

CSPOM a été conçu afin de simplifier la traduction/-reformulation des modèles, pour pouvoir adapter un modèle à un solveur. Un framework de reformulation facilement extensible est proposé dans ce but. Notre prototype supporte d'ores et déjà quatre langages de modélisation (XCSP, FlatZinc, une API Java et Scala très naturelle) et propose de plus quelques reformulateurs pouvant améliorer certains modèles naïfs.

1 Introduction

Le CSP Object Model (CSPOM) a été conçu à l'origine comme un analyseur syntaxique pour le langage de modélisation XCSP [4]. L'objectif était alors de réaliser une interface de programmation (*API*) bidirectionnelle entre un modèle objet et XCSP, dans le même esprit que la célèbre API Document Object Model (DOM) pour le langage XML. Une première version du modèle objet avait été présentée lors des 4es JFPC. Après quelques années de stabilité, la popularisation du langage de programmation Scala [7] et du langage de modélisation *MiniZinc* [6] nous a incité à retravailler complètement CSPOM. Le modèle objet a été étendu au niveau des variables afin de gérer les constantes et séquences de variables, mais simplifié au niveau des contraintes. Toutes les contraintes sont maintenant regroupées en une seule classe. D'autre part pour renforcer la modularité du système, nous avons développé un système de *compilateurs*. Ces compilateurs permettent de *reformuler* les contraintes d'un problème.

Le modèle lui-même de CSPOM est complètement indépendant de tout solveur, ce qui en fait une interface unifiée entre solveurs et langages de modélisation. En effet, outre une syntaxe différente, chaque langage de modélisation et surtout chaque solveur propose son propre ensemble de contraintes. Il suffit d'écrire les compilateurs qui permettront de passer d'un ensemble de contraintes à un autre, jusqu'à être compatible avec le solveur voulu. De plus, les compilateurs permettent d'améliorer un modèle naïf afin d'en simplifier considérablement la résolution.

À ce jour, une implémentation en Scala de CSPOM est disponible [9]. Elle inclut un analyseur syntaxique pour les langage *XCSP* 2.1 et *FlatZinc* 1.6. Le solveur *Concrete* utilise directement le modèle objet de CSPOM pour fournir une API utilisateur Java et Scala très naturelle, très proche de MiniZinc pour l'API Scala. Des ensembles de compilateurs sont proposés pour traduire les contraintes de ces différents langages vers les différentes contraintes supportées par le solveur *Concrete* [8].

2 Le modèle objet

Chaque variable du CSP est définie sur un domaine ; chaque contrainte implique un ensemble de variables. Un contrainte définit l'ensemble des affectations autorisées des variables qu'elle implique.

Le modèle objet de CSPOM présenté sur la figure 1 reflète cette définition classique des CSP. Pour rester indépendant des solveurs, le modèle objet de CSPOM ne fait aucune hypothèse sur l'ensemble des contraintes supportées par un solveur. La

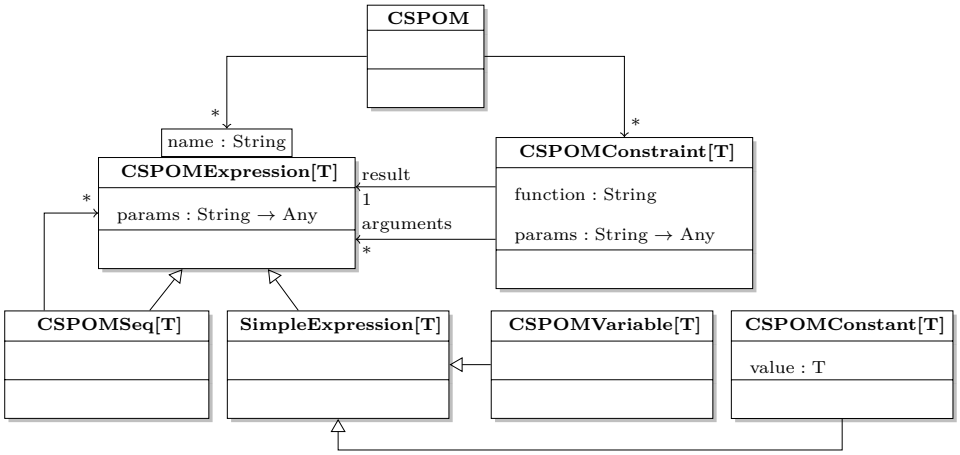


FIG. 1 : Extrait du modèle objet de CSPOM

classe CSPOMConstraint est donc réduite au minimum. Au niveau des variables cependant, la situation est plus complexe, afin d'apporter plus de souplesse aux développeurs de modèles. Les contraintes impliquent des CSPOMExpression. Les expressions peuvent être « simples » (constantes ou variables), ou encore des séquences d'expressions qui permettront de représenter vecteurs, tableaux, matrices, ensembles ordonnés… Certaines expressions peuvent être *nommées* au niveau du problème, ce qui permettra *in fine* d'interpréter les solutions trouvées par le solveur.

Une CSPOMConstraint comporte un attribut function qui sera utilisé par le solveur pour identifier le comportement de la contrainte (" alldifferent ", "weightedsum", "extension"…) ; il peut être complété par des paramètres. Par exemple, le solveur *Concrete* accepte des contraintes de type "extension" pour les contraintes *table*. La table elle-même est un paramètre. Chaque contrainte peut prendre un nombre arbitraire d'arguments (expressions), et une expression résultat : en effet, CSPOM considère chaque contrainte comme une fonction. Cela permet de décomposer facilement des contraintes complexes comme $X - Y \neq Z - T$ en contraintes « atomiques ». Le triplet (result , function , arguments) représentant une CSPOMConstraint (params est vide par défaut), la contrainte ci-avant, définie en notation infixe, sera naturellement décomposée en trois CSPOMConstraint par les analyseurs : $(V, \text{"sub"}, \langle X, Y\rangle)$, $(W, \text{"sub"}, \langle Z, T\rangle)$, $(\textbf{true}, \text{"ne"}, \langle V, W\rangle)$. Ce processus nécessite d'introduire des variables intermédiaires (V et W dans l'exemple précédent), aux domaines pour l'instant indéfinis. Des compilateurs pourront générer leurs domaines automatiquement (voir ci-après). Comme on peut le voir, les contraintes « usuelles », simplement posées sur une séquence de variables, ont la constante **true** comme résultat. C'est par exemple le cas de toutes les contraintes du langage *FlatZinc*. Placer une variable booléenne comme résultat permet de représenter facilement des contraintes réifiées.

Les constantes et les variables sont définies sur des sous-ensembles des types usuels des langages de programmation : booléens, nombres entiers ou flottants, voire une classe arbitraire si le langage et le solveur le supportent. Ce type est un paramètre de la classe (T sur le modèle ci-dessus). Pour les contraintes, le paramètre est celui de l'expression résultat. Ces types permettent aux compilateurs de détecter certaines erreurs dans le modèle, tout particulièrement quand on utilise directement CSPOM en Scala.

3 Langages supportés

L'API du solveur *Concrete* a été conçue pour exploiter le modèle et les fonctionnalités de CSPOM, et ainsi simplifier son utilisation et son développement. Elle permet d'écrire des modèles de contraintes de manière aussi naturelle qu'avec des langages dédiés comme *MiniZinc* [6] (cf figure 2). CSPOM rend la conception de telles API relativement simple pour d'autres solveurs. C'est particulièrement vrai quand il est possible d'utiliser le langage Scala, grâce à la possibilité de définir des opérateurs pour les CSPOMExpression (symboles $+$, $<$, etc.) et d'appeler des méthodes de conversion de manière implicite (valeurs en constantes, ajout transparent des contraintes au problème, etc.)

CSPOM implémente également des analyseurs syntaxiques pour les langages *XCSP* 2.1 et *FlatZinc*. *Concrete* utilise alors le framework de reformulation de CSPOM décrit ci-après pour convertir les contraintes définies dans ces langages en contraintes supportées par le solveur. Il n'y a presque rien à faire pour XCSP,

```
int: ticks;
int: maxt = ticks*ticks;
array[1..ticks] of var 1..maxt: t :: is_output;

constraint forall( i in 1..ticks-1 ) (
  t[i] < t[i+1] );

constraint forall ( i in 1..ticks, j in 1..ticks
  where i != j) (
  forall (k in 1..ticks, l in 1..ticks
    where k != l /\ (i != k \/ j != l) ) (
    t[i] - t[j] != t[k] - t[l] ));
```

(a) MiniZinc

```
val ticks = args(0).toInt
val max = ticks * ticks
CSPOM { implicit problem =>
  val variables = for (i <- 1 to ticks) yield
    interVar(1, max) as s"T$i"

  for (Seq(xi, xj) <- variables.sliding(2)) ctr(xi < xj)

  for (xi <- variables; xj <- variables if xi != xj;
       xk <- variables; xl <- variables if xk != xl &&
      (xi != xk || xj != xl) ) ctr(xi - xj !== xk - xl)
}
```

(b) API Scala de *Concrete*

FIG. 2 : Le problème de la règle de Golomb (mal) modélisé dans différents langages supportés par CSPOM

qui ne définit qu'un faible nombre de contraintes ; la décomposition en contraintes atomiques suffit. Il faut cependant générer les domaines des variables ajoutées au cours de la décomposition. Pour *FlatZinc* il faut généralement renommer les contraintes et restructurer les arguments (suivant les contraintes attendues par le solveur).

4 Le framework de reformulation

Dans CSPOM, un *compilateur* est une instance du type abstrait ConstraintCompiler. Il implémente une méthode de *matching* et une méthode de compilation proprement dite. La méthode de *matching* sert à identifier un « motif » qu'il sera possible de compiler. Le motif doit prendre une contrainte en argument principal : la méthode de *matching* sera appelée successivement pour chaque contrainte du problème. Le motif peut identifier quelques données qui seront transmises à la méthode de compilation. En pratique, il s'agit d'une fonction qui doit renvoyer les données identifiées, ou un symbole spécial si la contrainte n'est pas supportée par le compilateur.

Si le motif est validé, la méthode de compilation est appelée. Celle-ci sera généralement amenée à remplacer des contraintes ou variables par d'autres. En cas de modification, les contraintes au voisinage des contraintes ou variables ajoutées ou modifiées seront à nouveau contrôlées. CSPOM utilise pour cela une *file de compilation* inspirée de l'algorithme AC-3 [5].

Par exemple, le compilateur MergeSame permet de détecter quand deux contraintes affectent la même valeur à plusieurs variables, par exemple $X = Y - Z$ et $W = Y - Z$. Le motif est, pour une contrainte donnée, de rechercher toutes les contraintes ayant les mêmes fonction, arguments et paramètres que celle-ci. Ces contraintes sont alors remplacées par une égalité ($X = W$) lors de la compilation. Les méthodes de *matching* sont actuellement implémentés en Scala et utilisent les fonctionnalités du langage (notamment le *Pattern Matching*) pour effectuer des calculs relativement compliqués en peu de lignes de code. Certains motifs peuvent être complexes : une recherche de clique-max par recherche locale est implémentée dans l'un des compilateurs.

5 Performances de la reformulation

Pour évaluer la performance du compilateur de CSPOM, nous avons utilisé un modèle extrêmement naïf du problème de la règle de Golomb (n°6 de la CSPLIB, cf figure 2). Pour n coches à placer sur la règle, le modèle génère $O(n^4)$ contraintes de type $X - Y \neq Z - T$, la plupart strictement ou partiellement identiques par symétrie.

L'analyseur commence par décomposer ces contraintes en contraintes atomiques ($V = X - Y$…) Ensuite, le compilateur MergeSame détecte toutes les variables identiquement générées, permettant de réduire le total à $O(n^2)$ contraintes. Les variables identiques sont fusionnées par un autre compilateur. Enfin, la clique de contraintes $\neq$ obtenue est agrégée en une seule contrainte *all-different*. Comme le nombre de contraintes initial est relativement important (6 900 contraintes après décomposition pour le problème à 7 coches par exemple), le temps de compilation n'est pas négligeable, et peut s'avérer supérieur au temps de résolution du problème non simplifié pour les problèmes les plus faciles. Les résultats sont illustrés sur la figure 3, qui montre le temps de génération et de résolution du modèle brut, et après une phase de compilation permettant d'améliorer du modèle.

6 Travaux connexes

Les modèles définis en *MiniZinc* ont convertis en une version simplifiée du langage nommée *FlatZinc* avant d'être chargés par les solveurs. Le logiciel TAILOR effectue le même genre de travail pour le langage

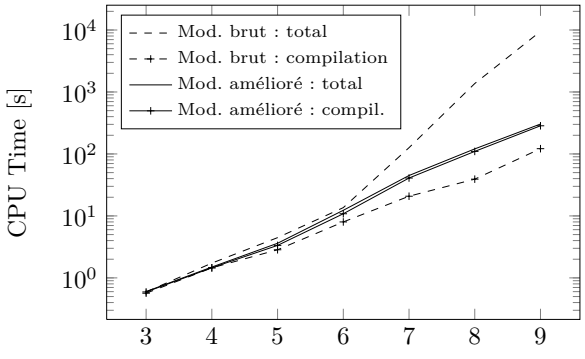


FIG. 3 : Performances sur le problème de la règle de Golomb (nombre de coches sur l'axe x).

ESSENCE' vers le format du solveur MINION [3]. La conversion consiste à « dérouler » les boucles, mais aussi à adapter, dans une certaine mesure, les modèles aux solveurs : il est possible de définir des prédicats qui seront décomposés en contraintes plus simples. Un grand nombre de prédicats sont fournis avec le langage. Par exemple, la célèbre contrainte *all-different* est convertie en une clique de contraintes $\neq$ si le solveur ne supporte pas la contrainte nativement. En utilisant un principe de recherche de motifs dans le graphe de contraintes, CSPOM permet d'implémenter le processus inverse. Bien que ce genre de reformulation soit beaucoup plus complexe, il est généralement possible d'améliorer certains modèles naïfs sans effectuer de calculs prohibitifs.

Le logiciel CGRASS [2] semble plus proche des fonctionnalités de CSPOM, mais n'est pas disponible. Aucune information n'est disponible sur la représentation des problèmes, les langages utilisés pour la définition des règles de reformulation. D'autre part, CSPOM implémente une file de compilation permettant de traiter des problèmes de taille supérieure.

Enfin, l'API JSR331 fournit une interface commune pour différents solveurs compatibles avec le langage Java [1]. Contrairement au JSR331, CSPOM se concentre sur le modèle et ne traite pas la définition des « buts » de résolution, qui sont à notre avis trop dépendants des solveurs. CSPOM nous semble plus simple à utiliser et à implémenter.

7 Perspectives

La première perspective est bien évidemment de développer des règles, notamment de détection de symétries. La définition d'un langage spécifique pour la reconnaissance de motifs pourrait également être intéressante. Les motifs employés pour détecter certaines propriétés du graphe de contrainte sont proches du problème de recherche de sous-graphes : l'utilisation d'algorithmes adaptés permettrait d'améliorer les performances de la phase de compilation. D'autre part, certaines reformulations complexes ne peuvent être obtenues qu'en *dégradant* en apparence le modèle, par exemple en ajoutant des contraintes redondantes, avant de le simplifier à nouveau. Ce genre de reformulation nécessiteraient d'une part de formaliser ce qu'est un « meilleur » modèle, et d'autre part l'emploi d'algorithmes de planification pour l'atteindre.

L'implémentation de CSPOM pour d'autres solveurs permettra de mieux valider son modèle objet. Il semblerait particulièrement intéressant d'utiliser CSPOM pour des solveurs supportant des types de données moins usuels (nombres flottants, classes arbitraires) ou utilisant d'autres paradigmes de résolution (SAT, SMT, mathématiques, incomplets, etc.) et d'autres langages de programmation.

Références

[1] J. FELDMAN et al. *JSR 331 : Constraint Programming API*. 2012.

[2] A.M. FRISCH, I. MIGUEL et T. WALSH. « CGRASS : A System for Transforming Constraint Satisfaction Problems ». Dans : *Recent Advances in Constraints, 15-30, LNCS 2627*. SpringerVerlag, 2002, p. 23–26.

[3] I.P. GENT, I. MIGUEL et A. RENDL. « Tailoring solver-independent constraint models : A case study with ESSENCE' and MINION ». Dans : *Proc. SARA'2007*. 2007, p. 18–21.

[4] C. LECOUTRE et O. ROUSSEL. « XML Representation of Constraint Networks, Version 2.1 ». Dans : *The Computing Research Repository* arXiv : 0902.2362v1 (2008).

[5] A.K. MACKWORTH. « Consistency in Networks of Relations ». Dans : *Artificial Intelligence* 8.1 (1977), p. 99–118.

[6] N. NETHERCOTE et al. « Minizinc : Towards a standard CP modelling language ». Dans : *Proc. CP'2007*. Sous la dir. de C. BESSIÈRE. LNCS 4741. Springer-Verlag, 2007, p. 529–543.

[7] M. ODERSKY et al. *Scala Language*. http ://scala-lang.org/. 2003–2014.

[8] J. VION. *Concrete : a CSP solving API for the JVM*. `http://github.com/concrete-cp`. 2006–2014.

[9] J. VION. *CSP Object Model*. `http://github.com/concrete-cp/cspom`. 2008–2014.

Oublier pour mieux régner : une courte étude expérimentale

Laurent Simon
Labri, Université Bordeaux
lsimon@labri.fr

Résumé

Les solveurs SAT basés sur l'apprentissage dirigé par les conflits sont construits autour d'un noyau dur de composants essentiels. Arrivent ensuite des composants secondaires, dont la stratégie d'oubli des clauses apprises. Cette dernière a pris de plus en plus d'importance ces dernières années mais, 5 ans après son introduction, ces travaux viennent d'être questionnés de manière fort pertinente il y a peu, notamment en prenant simplement en compte la taille des clauses. Malgré des résultats surprenants et parfois spectaculaires, nous avons jugé nécessaire de mettre en perspective les résultats obtenus. Nous montrons que la mesure introduite dans Glucose reste la mesure la plus efficace sur les instances industrielles classiques, tout en offrant une meilleur compréhension des limites et performances de la mesure basée sur la taille des clauses.

1 Introduction

Depuis l'avènement des solveurs SAT de type CDCL ([9, 3]), de nombreuses améliorations ont été proposées. Un apprentissage dirigé par les conflits de type First UIP ([6]), des structures de données paresseuses, des heuristiques hautement dynamiques (VSIDS), ainsi que diverses stratégies de redémarrages. En plus de ces quatre composants essentiels a été proposé dans Glucose [2] de prendre en compte la politique d'oubli des clauses apprises. L'introduction de la mesure appelée LBD (pour *Literal Block Distance* a permis au solveur `Glucose` de remporter de se hisser au tout premier plan des solveurs SAT sur les instances industrielles. Aujourd'hui pourtant, les solveurs n'ont plus les mêmes restrictions mémoire qu'en 2009 (accès au 64 bits et plusieurs dizaines de Go de mémoire). Le moment est peut être venu de remettre en cause la suprématie des mécanismes actuels. Ainsi, il a été proposé dans [4] de prendre plutôt en compte le nombre de niveaux de décision auxquels une clause a été propagée comme critère de qualité (en valeur absolue ou relativement au niveau courant). Plus récemment, les mêmes auteurs proposent dans [5] de prendre en compte simplement la longueur des clauses comme qualité (associé au niveau de propagation de la clause). Les performances obtenues, notamment en ajoutant un seuil au delà duquel un score aléatoire est utilisé, offrent des résultats assez spectaculaires sur les problèmes satisfiables utilisés dans la compétition 2013, avec un temps maximal relativement long (5000s). Sur les instances insatisfiable, les résultats affichés sont moins convaincants mais il n'en demeure pas moins que ce travail questionne de manière forte la pertinence des mesures introduites jusque là.

Cependant, comme on va le voir dans cette étude, les résultats cités précédemment doivent être pris avec une certaine précaution. Les données expérimentales rapportées dans [5] doivent être enrichies pour mieux appréhender l'intérêt et les limites de leur approche. Même si nos travaux viennent tempérer les résultats de ce travail, il est important de préciser combien l'intérêt de cette étude est indéniable et ne manque pas de poser de vraies questions. Cet article court (mais dense, et supposant la lecture de [5] pour une bonne compréhension) traite donc de la comparaison expérimentale entre les LBD et les principales politiques proposées dans [5]. Il s'agit de rapporter le travail personnel que nous avons accompli à la suite de la lecture de ce dernier article. Il nous a semblé pertinent de lui fournir un " *companion paper* ".

2 Travaux autour de la qualité des clauses

Il n'est pas possible de présenter tous les travaux effectués autour des solveurs CDCL et de la qua-

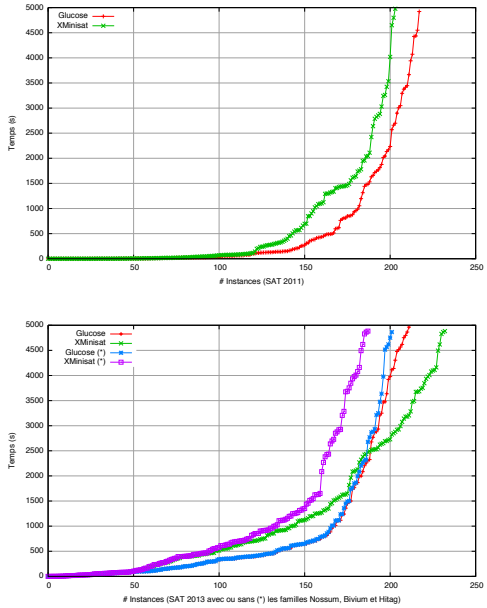


FIGURE 1 – Cactus des différents solveurs sur SAT 2011 (haut) et SAT 2013 (bas).

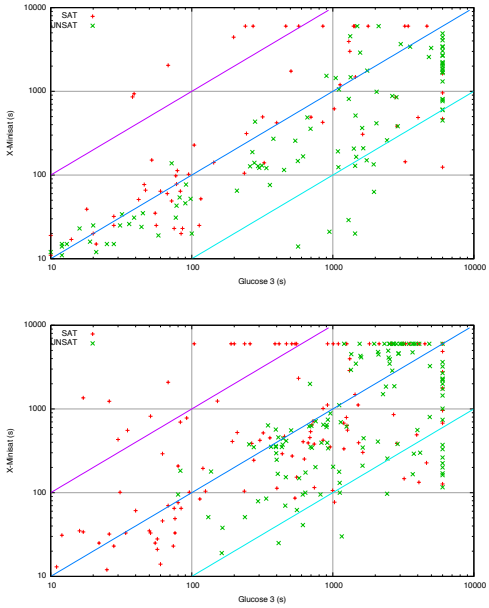


FIGURE 2 – Comparaison de `Glucose` (X) et `X-Minisat` (Y) sur les instances 2011 (haut) et 2013 (bas). Chaque point représente le temps de résolution d'une instance par les deux solveurs (X,Y).

lité des clauses. Le lecteur est supposé avoir une bonne connaissance des mécanismes décrits dans [3, 2], ainsi que [5], pour suivre correctement cet article. En quelques mots, cependant : dans [2], les auteurs proposent de considérer les blocs de propagations de littéraux comme des ensembles de littéraux " sémantiquement liés " (ces blocs caractérisent tous les littéraux impliqués après une nouvelle décision lors de la recherche). L'idée derrière `Glucose` étant d'ajouter les contraintes les plus fortes possibles entre ces blocs en donnant comme score à une clause le nombre de niveaux de décisions distincts auxquels sont affectés les littéraux la composant.

Principalement deux mesures sont proposées dans [5]. SBR(K) score une clause c ainsi : si $|c| < K$, son SBR(K) est $|c|$, sinon il est de $|c| + rand$, $rand$ étant un nombre aléatoire tiré une seule fois par clause. L'autre mesure, SIZE(K)D, propose de remplacer la valeur aléatoire par le niveau de décision où la clause a été propagée. Cette mesure est dynamiquement ajustée de manière décroissante, privilégiant donc les petites clauses, ou les clauses mettant en jeu des littéraux ayant été affectés ou propagés en haut de l'arbre de recherche, c'est à dire dépendant de variables qui, à un même moment de la recherche (au redémarrage ou lors de l'apprentissage d'une clause unaire) étaient *ensemble* les plus actives (donc vues souvent ensembles lors des récentes analyses de conflits). La mesure SIZE-D est la même mesure avec $K = 0$.

3 Expérimentations supplémentaires

Dans [5], seule est donnée la moyenne des temps CPU sur les instances ainsi que le nombre d'instances résolues. Or, les méthodes à base de SBR(K) offrent toutes des performances contrastées : si le nombre d'instances résolues est nettement amélioré, les performances moyennes sont fortement dégradées (par rapport à `Glucose`, `X-Minisat` peut prendre jusque 50% de temps supplémentaire). Est-ce dû à la résolution de quelques problèmes difficiles ? Est-ce une dégradation généralisée ? Nous avons testé 3 versions : `Glucose` 3.0, `X-Glucose` (SIZE(12)D) et `X-Minisat` (SIZE-D)) sur deux jeux d'instances. Ce dernier a été choisi comme représentant de [5] car ses performances sont proches des meilleurs et il ne nécessite pas de paramétrage sur le jeu d'instance (dépendant de K), ce qui permet une utilisation directe sur un plus grand jeu d'instances moins couteux en termes CPU.

Observons les figures *en cactus* figure 1. On voit tout d'abord que les observations (très nettes) sur SAT13 ne tiennent pas sur SAT11. Sur SAT13 maintenant, on observe deux choses. D'abord un croisement à 2000s de `Glucose` et `X-Minisat`. Ensuite, si on retire 3 familles d'instances de crypto (Nossum, Bivium et Hitag), la courbe de `Glucose` ne change pratiquement pas alors que celle de `X-Minisat` est fortement dégradée. `X-`

An.	Gagnant	Perdant	Score	DStd.	Prob.
13	`Glucose`	`X-Glucose`	12	1.21	0.89
13	`X-Glucose`	`X-Minisat`	29	2.26	0.99
13	`Glucose`	`X-Minisat`	18	1.37	0.92
11	`Glucose`	`X-Minisat`	45	4.62	1^*

TABLE 1 – Comparaison des solveurs par paires d'après [8]. * : 1 est l'arrondi de 0.999998. La première colonne indique le jeu de problèmes testés. La dernière colonne donne la probabilité que le solveur colonne 2 soit plus rapide que le solveur colonne 3.

`Minisat` est donc particulièrement adapté à la résolution de ces 3 séries de problèmes. Ces résultats sont confirmés par les figures en *log-log* figure 2. `Glucose` reste le plus efficace sur la majorité des instances UNSAT (celles justement pour lesquelles la mesure LBD a été introduite). Donc, même si `X-Minisat` offre des performances tout à fait impressionnantes en termes d'instances résolues sur les problèmes 2013, on voit que les conclusions très générales tirées dans [5] doivent être pris avec recul.

Pour étayer ces observations, nous avons utilisé l'approche proposée dans [8], dont le but est justement de tenter d'établir un classement indépendamment du temps maximal utilisé. Or, les résultats obtenus sont complètement inversés, comme le montre la table 1, même sur SAT 2013.

Par contre, si on prend la mesure PAR1 (utilisée par exemple pour sélectionner les meilleurs solveurs lors de la construction de `Satzilla`), on obtient bien le même classement que dans [5] : `Glucose` = 186K, `X-Glucose` = 183K et `X-Minisat` = 134K. En utilisant la mesure PAR10, cela enfonce encore plus le clou en faveur de `X-Minisat` : `Glucose` = 1401K, `X-Glucose` = 1398K et `X-Minisat` = 449K. La mesure PARX pénalise un solveur ne répondant pas sur une instance donnée par un temps de X*MaxCPU.

3.1 Particularités des instances 2013

Le danger des compétitions SAT se fait ressentir de manière aigüe lorsque l'on étudie la composition des instances 2013. Pour la première fois dans l'histoire de la compétition, près d'un tiers des instances viennent de problèmes de crypto. De plus, les 30 (10% des instances !) instances `Nossum` sont des instances de crypto générées pour pouvoir être résolue, en moyenne, en 5000s par `Minisat`. Rappelons que si on retire de l'ensemble des problèmes les instances `Nossum`, `Hitag` et `Bivium`, on observe une inversion de classement sur les problèmes restant de la compétition 2013. On notera aussi que dans [2], les auteurs avaient identifié les instances de crypto comme ardues pour leur approche (à la différence des instances BMC).

4 Discussion

Dans [5], les auteurs plaident à charges contre l'intérêt des LBD, de manière un peu surprenante. "*In our opinion, the performance of the LBD measure can be explained by the fact that it is really related to the size of the clauses. Indeed, we have* $2 \leq LBD(c) \leq |c|$. *The clause size is an upper bound of the LBD measure.*". Cette affirmation n'est cependant pas étayée par une quelconque étude expérimentale. Nous avons voulu la tester simplement. Sur un cluster Xeon E5420 avec 1200s de temps maximum et les instances 2011, `Glucose` résoud 183 instances. Si on substitue le score LBD par une valeur aléatoire entre 1 et la taille de la clause apprise (non mise à jour) alors `Glucose` ne résoud que 162 instances. Si on remplace le LBD par la taille de la clause, alors `Glucose` ne résoud que 171 instances (instances de la compétition 2011). Cette simple expérimentation semble contredire les auteurs (même si une expérimentation avec un temps plus long serait à mener). De plus, les auteurs discutent la pertinence de l'utilisation du LBD par rapport à la taille d'une clause. Ils défendent l'idée simple qu'une clause de taille n est plus importante qu'une clause de taille n' avec $n < n'$. Il est donc étonnant de constater que leur arguments ne tiennent pas dans leur approche dès que n dépasse K ou même pour la mesure SIZE-D (la valeur du niveau de propagation de la clause est fortement utilisé). Cependant, si la discussion à charge contre le LBD est surprenante, elle peut être comprise dans une certaine mesure : il est clair que `Glucose`, et nous pensons que c'était son rôle, a poussé à l'extrême l'utilisation du score LBD. Il s'agit aussi de défendre l'idée de solveurs CDCL s'éloignant de la vision plus ancienne basée sur DPLL.

Le succès de `Glucose` est-il dû à une simple corrélation avec la taille des clauses ? Un élément de solution a été déjà donné dans l'étude présentée [7]. Il a été mis à jour une corrélation forte entre la mesure LBD d'une clause et le nombre de communautés [1] qu'elle contient, ce qui tendrait à confirmer l'intuition initiale derrière `Glucose`. Notons aussi qu'en 2013, seuls 20% des problèmes étaient des problèmes de *Hardware*, des instances typiquement visées par `Glucose`.

4.1 Leçons à tirer

Les versions testées dans [5] sont autour de trois mesures : la taille de la clause, le niveau de décision auquel la clause a été propagée ainsi que l'utilisation de scores aléatoires. Suivant les cas, l'une ou l'autre semble être la plus efficace mais le message n'est pas clairement concluant : faut-il entièrement se fier à la mesure SBR(K) qui ne tient compte que de la taille et d'une mesure aléatoire ? Les performances sur les

instances satisfiables de cette dernière sont réellement remarquables.

De plus, il semble assez logique de questionner le lien entre la mesure BTL et la mesure LBD, surtout quand les stratégies de redémarrages suivent des lois Luby, donc sont extrêmement nombreuses. Nous avons mené une expérimentation en changeant la mesure Size(12)D-Glucose par la stratégie Size(12)L-Glucose, où le LBD est utilisé plutôt que le BTL et, même sur les instances 2013 (incluant les instances de cypto pour lesquels la notion de LBD ne semble pas pertinente), il est impossible de conclure entre l'une ou l'autre des approches. Ce dernier résultat, associé au fait que toutes les versions `X-Glucose` proposées accroissent significativement le temps moyen de résolution pour gagner entre 3 et 8 problèmes, devrait venir tempérer l'enthousiasme (à charge) des auteurs : "*This experiment give us a definitive illustration that clause size based measure is better than LBD*". D'autant plus que leurs conclusions ne tiennent plus dès qu'on ne se limite pas aux instances utilisées en 2013, ou que l'on retire de ces instances 3 familles de problèmes de crypto.

5 Conclusion

L'utilisation de la taille de la clause comme nouvelle mesure de pertinence, si elle était avérée, serait une surprise à plus d'un titre. Les expérimentations que l'on a mené montrent cependant que ce type de mesure n'est intéressant que pour les problèmes difficiles – mais pas trop – non structurées. Sur ces problèmes, la mesure LBD semble atteindre ses limites et laisse la place à toute une famille de nouvelles mesures. Cependant, on peut se demander comment ces techniques vont pouvoir passer à l'échelle pour la parallélisation. En effet, si une parallélisation de type portfolio est clairement prometteuse (puisque chaque coeur est immédiatement amélioré), le partage des clauses (qui nous semble être la seule voie intéressante de la parallélisation de solveurs SAT) risque d'être de nouveau confronté à une recherche de mesure de qualité : partager toutes les clauses inférieures à 15, par exemple, semble difficilement envisageable.

L'efficacité des solveurs SAT ne saurait être résumée par un seul paramètre, quel qu'il soit. On peut ainsi facilement trouver des sous familles mettant en cause les politiques de restarts classiques, ou de polarité de variables. Dans [5], les auteurs profitent d'un biais important dans la construction des problèmes 2013 pour offrir des conclusions un peu trop définitives par rapport à l'état de l'art. On pourrait par exemple étendre leur conclusion pour démontrer l'inefficacité de toutes les technologies à base de inprocessing (caractérisant `Lingeling`) ce qui serait contre productif.

En conclusion cependant, les travaux de [5] sont bel et bien intéressants, et à plus d'un titre. Ainsi, si on reprenait les règles des premières compétitions, leur approche ne franchirait pas la première phase de la compétition, étant significativement moins performante à 1000s. Les auteurs semblent donc proposer un mode opératoire des CDCL visant les instances difficiles et non structurées, ce qui n'avait pas été réellement attaqué jusqu'à présent (nécessité de grande consommation mémoire et d'un cluster de calcul performant). De plus, nous pensons que les travaux discutés dans [5] peuvent ouvrir la porte à une dichotomie entre les solveurs CDCL visant les instances SAT et les solveurs spécialisés dans les instances UNSAT. Ce simple résultat, en soit, représenterait un important progrès pour la communauté. Cependant, comme nous l'avons montré dans ce court article, les conclusions assez définitives des auteurs par rapport aux technologies à base de LBD doivent être lues avec beaucoup de précaution et un recul que veut permettre d'offrir cet article. Les technologies autour des LBD restent les plus efficaces sur les instances offrant une certaine structure.

Références

[1] Carlos Ansótegui, Jesús Giráldez-Cru, and Jordi Levy. The community structure of sat formulas. In *SAT*, volume 7317, pages 410–423. 2012.

[2] G. Audemard and L. Simon. Predicting learnt clauses quality in modern SAT solvers. In *IJCAI*, pages 399–404, 2009.

[3] N. Eén and N. Sörensson. MiniSat : A SAT solver with conflict-clause minimization. In *SAT*, 2005.

[4] L. Guo, S. Jabbour, and L. Sais. Stratégies d'élimination des clauses apprises dans les solveurs sat modernes. In *JFPC*, 2013.

[5] S. Jabbour, J. Lonlac, L. Sais, and Y. Sahli. Revisiting the learned clauses database reduction strategies. 1402.1956, 2014.

[6] J. Marques-Silva and K. Sakallah. GRASP – a new search algorithm for satisfiability. In *ICCAD*, pages 220–227, 1996.

[7] Zack Newsham, Gilles Audemard, Vijay Ganesh, Sebastian Fischmeister, and Laurent Simon. Impact of community structure on sat solver performance. In *Proceedings of SAT*, 2014.

[8] A. van Gelder. Careful ranking of multiple solvers with timeouts and ties. In *SAT*. 2011.

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

Recherche à voisinage large avec objectif variable. Une approche pragmatique pour résoudre les problèmes sur-contraints *.

Pierre Schaus

UCLouvain, ICTEAM
Place saint barbe 2
1348 Louvain-la-Neuve, Belgium
{pierre.schaus}@uclouvain.be

La programmation par contraintes permet de résoudre des problèmes d'optimisation combinatoire difficiles généralement en utilisant une recherche par séparation et évaluation en explorant l'arbre de recherche en profondeur d'abord. Malheureusement, cette approche rencontre rapidement des difficultés de passage à l'échelle lorsque la fonction objectif est en réalité une somme de variables à minimiser. Comme expliqué dans [3], l'origine de ce problème est principalement le faible filtrage (consistance aux bornes) des contraintes de sommes. Afin de réduire l'impact négatif du faible filtrage induit par une fonction objectif de type somme de variables, nous suggérons d'étendre le cadre classique de la recherche à voisinage large (LNS) [7] en lui permettant de changer dynamiquement la fonction objectif à chaque redémarrage. La but de cette approche, que nous appelons VO-LNS, est de résoudre un problème réel de création d'horaires de cours qui est sur-contraint mais qui peut néanmoins être relâché en introduisant des variables de violation tel que proposé dans [2]. Nos expériences montrent que cette approche offre les avantages principaux suivants :

1. un filtrage plus fort sur chaque terme permettant d'améliorer la vitesse de LNS dans la découverte de solutions de bonnes qualités et
2. un meilleur contrôle afin d'équilibrer les termes qui composent la fonction objectif.

Ce dernier point est particulièrement intéressant dans les problèmes sur-contraints où l'on souhaite minimiser la somme des violations tout en gardant un certain équilibre sur l'ensemble des violations.

*Ceci est un court résumé de l'article [5].

Dans le cadre VO-LNS, un problème d'otimisation est décrit comme suit :

$$\begin{array}{ll} \text{Optimize} & obj = (obj_1, obj_2, \ldots, obj_m) \\ \text{Subject to} & constraints \end{array}$$

A la différence d'une formulation classique, plusieurs objectifs peuvent être spécifiés et chaque objectif doit être optimisé le mieux possible. Nous ne sommes donc pas intéressés par une approche multi-critères afin de collecter un front de Pareto tel que proposé dans [6]. L'objectif peut soit être une maximisation soit être une minimisation contenant une variable. L'utilisateur peut librement configurer chaque objectif dans un des trois modes suivants pour la recherche par séparation et évaluation :

1. *Filtrage inactif* : Cela signifie que cet objectif n'a pas d'impact du tout.
2. *Filtrage faible* : Lorsqu'une solution est trouvée durant la recherche par séparation et évaluation, la borne sur l'objectif est mise à jour de sorte que la prochaine solution trouvée devra être au moins aussi bonne que la précédente du point de vue de cet objectif.
3. *Filtrage fort* : Lorsqu'une solution est trouvée durant la recherche par séparation et évaluation, la borne de l'objectif est mise à jour de sorte que la prochaine solution trouvée devra être strictement meilleure du point de vue de cet objectif.

La Figure 1 montre à titre d'exemple l'évolution de la violation totale sur un problème sur-contraint de création d'horaire comportant une trentaine d'objectifs représentant chacun la violation d'une contrainte

soft-gcc [8, 9, 4]. La configuration A résulte d'un LNS standard avec relaxation aléatoire de variables. La configuration B est un LNS standard avec relaxation dédiée des variables. La configuration C utilise la même relaxation que B mais aussi VO-LNS afin de configurer le mode de l'objectif de moindre qualité dans la solution courante avec un filtrage fort tout en maintenant les autres objectifs avec un filtrage faible. Comme le montre les résultats, VO-LNS, grâce à l'augmentation du filtrage, permet d'atteindre une violation totale en un nombre nettement réduit de redémarrages (restarts) LNS.

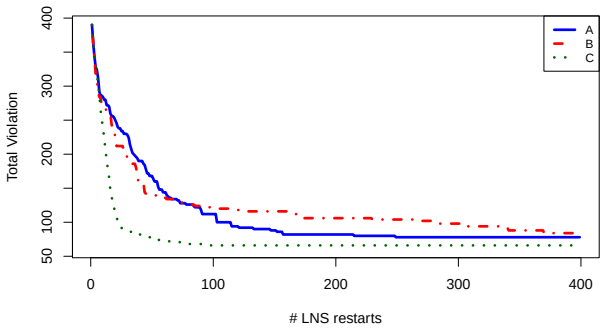


FIGURE 1 – Evolution de la violation totale au cours des redémarrages LNS sur un problème de création d'horaires avec trois configurations LNS différentes, C étant la configuration utilisant VO-LNS.

VO-LNS permet d'obtenir également un meilleur contrôle sur l'importance des sous-objectifs à minimiser en les sélectionnant par exemple plus ou moins fréquemment dans un filtrage faible ou fort. Un autre avantage de VO-LNS est qu'il ne nécessite quasiment pas de changer les solvers CP existants offrant déjà la fonctionnalité LNS. VO-LNS n'est pas nécessairement approprié pour tous types de problèmes. En particulier si la tension est trop forte entre des sous-objectifs, VO-LNS peut ne pas aider et il pourrait même empêcher LNS d'échapper aux optima locaux de par son approche trop agressive sur le filtrage. Néanmoins pour de nombreux problèmes, la tension entre les sous-objectifs n'est pas si forte et VO-LNS peut s'avérer d'une grande aide. Nous pensons que c'est souvent le cas sur les problèmes légèrement sur-contraints où l'on cherche à minimiser la somme de la violation de petites contraintes ayant un impact limité. Nous invitons le lecteur intéressé d'en savoir plus sur cette approche à se référer à l'article complet [5]. Une implémentation open source de VO-LNS est disponible dans OscaR 1.0.0 [1].

Références

[1] OscaR Team. OscaR : Scala in OR, 2012. Available from `https://bitbucket.org/oscarlib/oscar`.

[2] T. Petit, J.C. Régin, and C. Bessière. Meta-constraints on violations for over constrained problems. In *Tools with Artificial Intelligence, 2000. ICTAI 2000. Proceedings. 12th IEEE International Conference on*, pages 358–365. IEEE, 2000.

[3] J.C. Régin and T. Petit. The objective sum constraint. *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, pages 190–195, 2011.

[4] P. Schaus, P. Van Hentenryck, and A. Zanarini. Revisiting the soft global cardinality constraint. *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, pages 307–312, 2010.

[5] Pierre Schaus. Variable objective large neighborhood search : A practical approach to solve over-constrained problems. In *IEEE International Conference on Tools with Artificial Intelligence (ICTAI)-2013*, 2013.

[6] Pierre Schaus and Renaud Hartert. Multi-objective large neighborhood search. In *Principles and Practice of Constraint Programming*, pages 611–627. Springer, 2013.

[7] P. Shaw. Using constraint programming and local search methods to solve vehicle routing problems. *Principles and Practice of Constraint Programming-CP98*, pages 417–431, 1998.

[8] W.J. van Hoeve, G. Pesant, and L.M. Rousseau. On global warming : Flow-based soft global constraints. *Journal of Heuristics*, 12(4) :347–373, 2006.

[9] A. Zanarini, M. Milano, and G. Pesant. Improved algorithm for the soft global cardinality constraint. *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, pages 288–299, 2006.

Recherche à voisinage large guidée par l'impact sur le cout *

Michele Lombardi[1] **et Pierre Schaus**[2]

[1] DISI, University of Bologna
[2] ICTEAM, Université Catholique de Louvain, Belgium

michele.lombardi2@unibo.it pierre.schaus@uclouvain.be

Dans les recherches à voisinage large (LNS) [10], un problème d'optimisation est résolu en explorant de manière successive (via l'exploration d'un arbre de recherche) un voisinage autour d'une solution courante. Lorsque qu'une solution meilleure est trouvée, celle-ci remplace la solution courante. Les approches LNS permettent généralement à la programmation par contrainte de passer à l'échelle sur les problèmes réels de grandes tailles, pour autant qu'une bonne heuristique de création de voisinage existe. Malheureusement, la création d'une telle heuristique de voisinage est tout un art et dans de nombreux cas il est préférable d'utiliser une bonne connaissance du problème pour faire mieux qu'une simple relaxation aléatoire des variables.

Récemment certains chercheurs ont émis l'idée d'inclure dans le voisinage les variables qui impactent le plus sur le coût de la solution courante [6, 2]. Ces approches sont malheureusement assez spécifiques au domaine ou nécessitent de profondes modifications des solveurs existants. La méthode PGLNS [8] n'est pas orientée coût, mais orientée propagation et à l'avantage d'être indépendante du problème à résoudre. Cette méthode tente de découvrir les variables liées entre elles afin de définir un voisinage basé sur ces liens.

Notre contribution est en quelque sorte un mixe de ces idées. Comme dans [6], nous utilisons la propagation du solveur pour mesurer l'impact sur le coût. Nous reconnaissons également comme dans [2] que les variables affectant le plus le cout doivent être relâchées préférentiellement. Finalement, comme pour PGLNS [8], notre approche est indépendante du problème et ne nécessite pas de désactiver la propagation de certaines contraintes lors du calcul d'impact sur le coût.

Dans ce travail, nous proposons de se reposer sur la propagation des contraintes et les fonctionnalités de base des solveurs afin de créer un ensemble d'heuristiques de voisinage simples, basées sur le cout et totalement indépendantes du problème à résoudre. Ces techniques sont appliquées aux problèmes de Steel Mill Slab illustrant la supériorité de l'approche proposée par rapport à des relaxations aléatoires.

Formellement, définissions $P = \langle z, X, D, C\rangle$ un problème d'optimisation sous contraintes, avec X un ensemble de variables, D est l'ensemble des domaines des variables (avec D_i le domaine de x_i), et z est la variable coût. Sans perte de généralité, nous supposons que z a un domaine initial $[-\infty, \infty]$ et doit être minimisée. L'ensemble C contient les contraintes du problème. Chaque contrainte c_i est définie sur un sous ensemble de variables $S(c_i)$, sa *portée*. La portée peut inclure des variables de X ainsi que la variable z.

Nous considérons une affectation (partielle) τ comme une contrainte particulière qui force chaque variable x_i dans sa portée $S(\tau)$ à prendre une valeur spécifique $v_i = \tau(x_i)$. Une affectation τ est une solution si $S(\tau) = X$ et que le problème $P_\tau = \langle z, X, D, C \cup \tau\rangle$ est cohérent (aucune de ses contraintes ne détecte d'infaisabilité). Nous utilisons la notation spéciale σ pour dénoter les solutions.

A chaque redémarrage LNS, nous commençons depuis une solution σ, ensuite nous sélectionnons un sous ensemble X_R des variables à relâcher et nous construisons une affectation partielle τ telle que :

— la portée inclut les variables à fixer, i.e. $S(\tau) = X \setminus X_R$
— $\tau(x_i) = \sigma(x_i)$ pour chaque $x_i \in S(\tau)$

*Ceci est un court résumé de l'article [4].

ensuite nous tentons trouver une solution pour P_τ avec un coût meilleur. Plus précisément, la notation $lb_\tau(x_i)$ représente la borne inférieure du domaine de x_i après la propagation jusqu'au point fixe sur P_τ. Ensuite une solution σ' a un coût meilleur que σ ssi $lb_{\sigma'}(z) < lb_\sigma(z)$.

Le choix des variables X_R à relâcher est crucial pour l'efficacité d'une approche LNS. Notre méthode repose sur une métrique basée sur la variation de la borne inférieure de la variable de coût [1]. A la différence de [6], nous collectons ces variations en rejouant incrémentallement la solution courante en et en changeant l'ordre dans lequel cette solution est rejouée. Nous effectuons en quelque sorte un *plongeons* (diving). De manière plus précise :

Definition 0.1. *Soit π une permutation des variables dans X et k est la position de x_i dans π. Alors l'*impact-coût *de x_i par rapport à une solution σ est la quantité :*

$$\mathcal{I}^z(x_i, \sigma, \pi) = lb_{\tau_{\pi,k}}(z) - lb_{\tau_{\pi,k-1}}(z) \quad (1)$$

où

$$S(\tau_{\pi,k}) = \{x_{\pi_j} \mid j = 0...k\} \quad (2)$$
$$\tau_{\pi,k}(x_i) = \sigma(x_i) \quad \forall x_i \in S(\tau_{\pi,k}) \quad (3)$$

i.e., $\tau_{\pi,k}$ force les $k+1$ premières variables dans π à prendre la valeur qu'elles ont dans σ.

En d'autres mots, notre mesure d'impact est simplement la variation de la borne inférieure enregistrée lors de la k-ième affectation pendant que la solution courante σ est rejouée dans l'ordre spécifié par π. Il est possible d'agréger le coût des impacts sur un ensemble de plongeons Π en les moyennant :

$$\mathcal{I}^z(x_i, \sigma, \Pi) = \frac{1}{|\Pi|} \sum_{\pi \in \Pi} \mathcal{I}^z(x_i, \sigma, \pi) \quad (4)$$

Une mesure indépendante de la permutation serait d'agréger le coût pour toutes les permutations possibles. Cela étant prohibitif à calculer, nous proposons d'utiliser une approximation à l'aide de l'impact moyenné sur un nombre fini de plongeons. Le nombre de plongeons à effectuer ainsi que le choix des permutation π pour chaque sont les décisions qui doivent être prises pour la définition de notre heuristique de recherche. Nous avons expérimenté notamment la stratégie suivante :

— Pour la fréquence de plongeons : 1) n plongeons par itérations LNS et 2) plonger toutes les n itérations LNS (afin de limiter le sur-coût de calcul).

1. Il y a une certaine similarité avec l'idée des pseudo-costs pour le MIP [1].

Algorithm 1 Probabilité basée sur l'impact coût

1: Affectation d'une score s_i à chaque variable (voir Equation 5)
2: let $r = \sum_{x_i} s_i$
3: **while** pas suffisamment de variables sélectionnées pour la relaxation **do**
4: choisir une valeur aléatoire v in $[0, r]$
5: **for all** x_i non sélectionné **do**
6: $v = v - s_i$
7: **if** $v \leq 0$ **then**
8: $r = r - s_i$
9: sélectionner x_i pour la relaxation et continuer à la ligne 2

— Pour la choix des permutations : 1) des permutations aléatoires uniformes et 2) des permutations construites dans l'ordre décroissant de l'impact (les variables sont triées dans π dans l'ordre décroissant de leur impact afin d'essayer de répartir le coût des variations).

Etant donné que le coût des impacts dépend de la solution courante, les impacts accumulés sont réinitialisés lorsqu'une nouvelle solution meilleure est découverte. Si aucune solution améliorante n'est trouvée, chaque plongeon augmente le coût agrégé des impacts, de sorte que l'estimation converge vers la moyenne réelle.

Nous avons expérimenté différentes stratégies de sélection sur base de cette information. La stratégie obtenant les meilleurs résultats exploite l'impact-coût tout en biaisant la probabilité d'une variable d'être *relâchée*. La méthode est décrite dans l'Algorithme 1 et consiste à tirer un nombre fixé de variables depuis X, sans replacement. Les probabilités de tirage sont données par un score (à la ligne 1), qui dans notre cas est une combinaison convexe de l'impact coût et d'une quantité uniforme :

$$s_i = \alpha \cdot \mathcal{I}^z(x_i, \sigma, \Pi) + (1-\alpha) \cdot \frac{1}{|X|} \sum_{x_j \in X} \mathcal{I}^z(x_j, \sigma, \Pi) \quad (5)$$

La présence d'un terme uniforme garanti que même les variables ayant un impact-coût null ont une chance d'être relâchées. La stratégie n'utilise qu'un seul paramètre additionnel $\alpha \in [0, 1]$ (avec $\alpha = 0$ correspondant à une sélection purement aléatoire). Dans nos expériences nous utilisons $\alpha = 0.5$. Nous plongeons toutes les 10 tentatives LNS (échouées) et également à chaque fois qu'une solution améliorante est trouvée.

La capacité à diversifier est la raison pour laquelle une relaxation purement aléatoire donne de très bons résultats sur certains problèmes. Par exemple, sur le problème du Steel Mill Slab, une relaxation purement aléatoire est la meilleure stratégie dans [3, 9]. Mairy et al. arrivent à la même constatation dans [5] concluant

que leur stratégie basée sur l'apprentissage par renforcement n'est pas meilleure que la relaxation aléatoire sur les problèmes de séquencement de voitures.

Nous avons utilisé les instances les plus couramment utilisées pour les problèmes de Steel Mill Slab dans la littérature [2], nous avons sélectionné les instances les plus difficiles avec 2,3,4 and 5 capacités de moules (80 instances au total) [9]. Nous avons limité le nombre d'itérations LNS à 1,000, de sorte que la meilleure solution est suffisamment stable pour les trois stratégies de relaxation expérimentées. La taille du voisinage est de 5 variables relâchées et chaque itération LNS est arrêtée après au plus 50 échecs, comme dans [9]. Toutes les expériences sont réalisées avec le solveur OscaR [7]. Les détails des résultats sont disponibles dans [4]. En résumé, la technique proposée de relaxation basée sur l'impact-coût domine les relaxations aléatoires ainsi que la relaxation Reversed PGLNS [8] sur la plupart des instances. De manière étonnante PGLNS est même moins bon que la relaxation aléatoire sur ce problème [3]. Pour toutes les instances nous avons utilisé un test de Student's t-test pour vérifier si l'amélioration est statistiquement significative. Pour 75 des 80 instances, la relaxation basée sur le coût est bien significativement meilleure au seuil de confiance 5%. Nous invitons le lecteur intéressé d'en savoir plus sur cette approche et les résultats à se référer à l'article complet [4].

Références

[1] M Benichou, JM Gauthier, P Girodet, G Hentges, G Ribiere, and O Vincent. Experiments in mixed-integer linear programming. *Mathematical Programming*, 1(1) :76–94, 1971.

[2] Tom Carchrae and J. Christopher Beck. Principles for the design of large neighborhood search. In *Journal of Mathematical Modelling and Algorithms*, volume 8, pages 245–270, 2009.

[3] Antoine Gargani and Philippe Refalo. An efficient model and strategy for the steel mill slab design problem. In *Principles and Practice of Constraint Programming–CP 2007*, pages 77–89. Springer, 2007.

[4] Michele Lombardi and Pierre Schaus. Cost impact guided lns. In *International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, CPAIOR2014*, 2014.

2. Celles-ci sont disponibles à l'adresse `http://becool.info.ucl.ac.be/steelmillslab`.

3. Cela peut être du à une différence dans notre implémentation par rapport à celle de [8].

[5] Jean-Baptiste Mairy. Reinforced adaptive large neighborhood search. In *The Seventeenth International Conference on Principles and Practice of Constraint Programming (CP 2011)*, page 55, 2011.

[6] Jean-Baptiste Mairy, Pierre Schaus, and Yves Deville. Generic adaptive heuristics for large neighborhood search. *Seventh International Workshop on Local Search Techniques in Constraint Satisfaction (LSCS2010). A Satellite Workshop of CP*, 2010.

[7] OscaR Team. OscaR : Scala in OR, 2012. Available from `https://bitbucket.org/oscarlib/oscar`.

[8] Laurent Perron, Paul Shaw, and Vincent Furnon. Propagation guided large neighborhood search. In Mark Wallace, editor, *Principles and Practice of Constraint Programming - CP 2004*, volume 3258 of *Lecture Notes in Computer Science*, pages 468–481. Springer, 2004.

[9] Pierre Schaus, Pascal Van Hentenryck, Jean-Noël Monette, Carleton Coffrin, Laurent Michel, and Yves Deville. Solving steel mill slab problems with constraint-based techniques : Cp, lns, and cbls. *Constraints*, 16(2) :125–147, 2011.

[10] Paul Shaw. Using constraint programming and local search methods to solve vehicle routing problems. In *Principles and Practice of Constraint Programming - CP98*, pages 417–431. Springer, 1998.

Résolution du problème de routage quorumcast en programmation entière (résumé étendu)

Quoc Trung BUI[1] **Quang Dung PHAM**[2] **Yves DEVILLE**[1]

[1] ICTEAM, Université catholique de Louvain, Belgique
[2] SoICT, Hanoi University of Science and Technology, Vietnam
{quoc.bui, yves.deville}@uclouvain.be dungpq@soict.hust.edu.vn

Résumé

Cet article est un résumé de l'article publié à CPAIOR 2014 [3]. Le problème de routage de quorumcast est une généralisation de la multidiffusion qui se pose dans de nombreuses applications distribuées. Il consiste à trouver un arbre de coût minimum qui couvre un nœud source et au moins q de m nœuds spécifiés dans un graphe non dirigé. Dans cet article, nous résolvons ce problème en programmation entière. Les résultats expérimentaux montrent que nos quatre approches sont meilleures que l'état de l'art. Une analyse de sensibilité est également effectuée sur les valeurs de q et m.

1 Introduction

Soit un graphe non dirigé $G = (V, E, c)$, avec V, E, respectivement, l'ensemble des nœuds et l'ensemble des arêtes. A chaque arête $(i, j) \in E$ est associée un coût positif $c_{ij} \in \mathcal{R}^+$. Etant donné un ensemble de nœuds de multidiffusion $S \subseteq V$, une valeur entière $q \leq |S|$ et un nœud racine r (sans perte de généralité, on peut supposer que $r \in S$), l'objectif du problème de routage de quorumcast (QRP) est de trouver un arbre de coût minimum T qui couvre r et au moins q nœuds de S [2, 5, 4, 7, 6]. QRP est un problème NP-difficile qui apparaît dans de nombreuses applications distribuées telles que la synchronisation et la mise à jour d'une ressource répliquée. Pour résoudre QRP, différentes approches ont été proposées [2, 4, 7, 5, 6]. Les approches développées dans [6] représentent l'état de l'art, tant en méthode incomplète (recherche locale) que complète (programmation par contraintes).

Contributions Dans cet article, nous proposons quatre formulations mathématiques pour QRP et nous les utilisons pour résoudre QRP en programmation entière. Ces approches sont meilleures que les méthodes exactes de l'état de l'art. En outre, grâce aux résultats expérimentaux, nous analysons l'effet des valeurs q et $|S|$ sur les performances des approches proposées.

2 Modèles mathématiques

Dans cette section, nous proposons quatre modèles mathématiques pour QRP. La première formulation est basée directement sur le graphe non dirigé G, tandis que les autres sont basées sur le graphe dirigé correspondant, formé en remplaçant chaque arête de G par deux arcs opposés avec le même coût que l'arête d'origine.

Ces modèles peuvent exploiter les propriétés des solutions de QRP. Soit T une solution de $QRP(q, m)$ sur un graphe G. On peut facilement montrer que (1) toutes les feuilles de T sont des nœuds de multidiffusion [5], et (2) T s'étend sur exactement q nœuds de multidiffusion.

Tous les modèles utilisent les variables binaires x_{ij} indiquant si l'arête (i, j) est dans la solution T (dans le graphe non dirigé, nous utilisons la convention $i < j$). L'objectif des modèles est de minimiser $\sum_{(i,j)\in E} c_{ij}x_{ij}$.

2.1 Formulation naturelle : Model 1

Ce modèle introduit des variables binaires y_i indiquant si le nœud i est dans T.

$$\sum_{i,j\in C:(i,j)\in E} x_{ij} \leq |C|-1, \forall C \subset V, 2 \leq |C| \leq |V|-1 \quad (1a)$$

$$\sum_{(i,j)\in E} x_{ij} + \sum_{(j,i)\in E} x_{ji} \geq y_i, \forall i \in V \quad (1b)$$

$$\sum_{(i,j)\in E} x_{ij} + \sum_{(j,i)\in E} x_{ji} \leq (|V|-1)y_i, \forall i \in V \quad (1c)$$

$$1 + \sum_{(i,j)\in E} x_{ij} = \sum_{v\in V} y_v \quad (1d)$$

$$y_r = 1 \quad (1e)$$

$$\sum_{v\in S} y_v = q \quad (1f)$$

$$x_{ij} \in \{0,1\}, \forall (i,j) \in E \quad (1g)$$

$$y_i \in \{0,1\}, \forall i \in V \quad (1h)$$

2.2 Formulation basée sur la multi-flux : Model 2

Ce modèle introduit des variables $y_{ij}^k \in \mathcal{R}^+$ pour mesurer le flux, à travers l'arc $(i,j) \in A$, à partir de la racine r jusqu'au nœud $k \in V \setminus \{r\}$.

$$\sum_{(r,i)\in A} (y_{ri}^k - y_{ir}^k) \leq 1, \forall k \in V \quad (2a)$$

$$\sum_{k\in S, k\neq r, (r,i)\in A} (y_{ri}^k - y_{ir}^k) = q-1 \quad (2b)$$

$$\sum_{(k,i)\in A} (y_{ki}^k - y_{ik}^k) \geq -1, \forall k \in V \quad (2c)$$

$$\sum_{k\in S, k\neq r, (k,i)\in A} (y_{ki}^k - y_{ik}^k) = -(q-1) \quad (2d)$$

$$\sum_{(j,i)\in A} (y_{ij}^k - y_{ji}^k) = 0, \forall k \in V, i \in V \setminus \{k,r\} \quad (2e)$$

$$y_{ij}^k \leq x_{ij}, \forall (i,j) \in A, \forall k \in V \cup \{r\} \quad (2f)$$

$$y_{ij}^k \geq 0, \forall (i,j) \in A, \forall k \in V \cup \{r\} \quad (2g)$$

$$x_{ij} \in \{0,1\}, \forall (i,j) \in V \quad (2h)$$

2.3 Formulation classique : Model 3

Ici, nous proposons une formulation basée sur le graphe dirigé $G' = (V, A)$.

$$x_{ir} = 0, \forall (i,r) \in A \quad (3a)$$

$$\sum_{(r,i)\in A} x_{ri} \geq 1 \quad (3b)$$

$$\sum_{u\notin C, v\in C, (u,v)\in A} x_{uv} \geq \sum_{(j,i)\in A} x_{ji},$$
$$\forall C \subset V, 2 \leq |C| \leq |V|-1, \forall i \in C, r \notin C \quad (3c)$$

$$\sum_{(i,j)\in A} x_{ij} \leq 1, \forall j \in V \quad (3d)$$

$$\sum_{i\in S, i\neq r, (j,i)\in A} x_{ji} = q-1 \quad (3e)$$

$$x_{ij} \in \{0,1\}, \forall (i,j) \in A \quad (3f)$$

2.4 Formulation de Miller-Tucker-Zemlin : Model 4

Dans cette section, nous proposons une formulation utilisant les contraintes de Miller-Tucker-Zemlin sur le graphe dirigé $G' = (V, A)$. Ce modèle introduit des variables t_i telles que que $t_i < t_j$ si $(i,j) \in T$ et des variables p_i pour indiquer l'appartenance du nœud i dans T.

$$p_r = 1 \quad (4a)$$

$$x_{ir} = 0, \forall (i,r) \in A \quad (4b)$$

$$\sum_{(r,i)\in A} x_{ri} \geq 1 \quad (4c)$$

$$\sum_{(i,j)\in A} x_{ij} = p_j, \forall j \in V \setminus \{r\} \quad (4d)$$

$$x_{ij} \leq p_i, \forall (i,j) \in A \quad (4e)$$

$$|V| x_{ij} + t_i + 1 \leq t_j + |V|, \forall (i,j) \in A \quad (4f)$$

$$1 + \sum_{(i,j)\in A} x_{ij} = \sum_{v\in V} p_v \quad (4g)$$

$$\sum_{i\in S} p_i = q \quad (4h)$$

$$x_{ij} \in \{0,1\}, \forall (i,j) \in A \quad (4i)$$

$$p_i \in \{0,1\}, \forall i \in V \quad (4j)$$

$$t_i \in \{1 \dots |V|\}, \forall i \in V \quad (4k)$$

3 Résolution du problème de routage de quorumcast en programmation entière

Dans cette section, nous proposons cinq approches différentes, basées sur les modèles ci-dessus, pour résoudre QRP. Les modèles 2 et 4 ont un nombre polynomial de variables et de contraintes. Ils peuvent être directement utilisés dans un solveur de MIP (CPLEX). Ces approches seront désignés *Mod2_B&B* et *Mod4_B&B*. Les modèles 1 et 3 ont un nombre exponentiel de contraintes. Les contraintes sont relâchées, et différentes approches de Branch& Cut sont utilisées.

Approche contraintes paresseuses Cette approche est appliquée aux modèles 1 et 3, où les contraintes de connectivité (1a) et (3c) sont considérées comme des contraintes paresseuse. Pour plus détail sur cette approche, cfr [1, 3]. Les deux approches correspondantes seront désignées *Mod1_B&C_lazy* et *Mod3_B&C_lazy*.

Approche dynamique de séparation de contrainte Cette approche peut être appliquée au modèle 3, où les contraintes de connectivité (3c) sont dynamiquement séparées [1, 3]. Cette approche sera dénotée par *Mod3_B&C_dyn*.

Prétraitement Plusieurs réductions ont été proposées pour le problème de l'arbreainis que pour d'autres

Class	Approach	%*opt*	$\overline{I}$	$\overline{N}$	$\overline{C}$	$\overline{T}$
C1	*CP*	97.1	na.	na.	na.	80.47
	Mod1_B&C_lazy	100	441.7	50.43	6.58	0.57
	Mod2_B&B	100	2159	1.38	na.	1.06
	Mod3_B&C_lazy	100	398.8	77.51	81.42	0.35
	Mod3_B&C_dyn	100	308.2	3.69	158.1	1.94
	Mod4_B&B	100	506.2	55.9	na.	0.64
C2	*CP*	0.08	na.	na.	na.	9.74
	Mod1_B&C_lazy	78.6	92804	8507	1110	150.9
	Mod2_B&B	60.2	66716	7.05	na.	318.6
	Mod3_B&C_lazy	94.4	30938	2312	1077	97.6
	Mod3_B&C_dyn	77.2	8408	54.64	6089	153.0
	Mod4_B&B	95.2	50327	6138	na.	92.8

FIGURE 1 – Un résumé des résultats de calcul pour deux classes d'instances

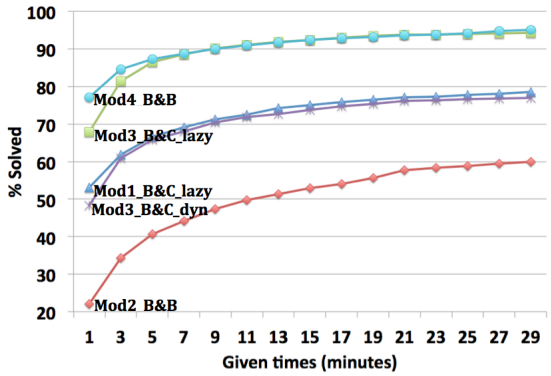


FIGURE 2 – Pourcentage d'instances résolues dans **C2** dans les temps donnés

problèmes liés. Dans le prétraitement de QRP, la réduction *vérifier nœuds inutiles* est intéressante. Elle est réalisée sur le graphe non dirigé G. Si un nœud n'est pas un nœud de multidiffusion, et a un degré de 1, alors ce nœud (et son arête) peuvent être enlevés du graphe G. Si un nœud v n'est pas un nœud multidiffusion et a exactement deux voisins u et w, alors le nœud v et les arêtes (u, v) et (v, w) peuvent être enlevés. Si il existe une arête (u, w) avec un coût c_{uw}, ce coût est mis à jour à $min(c_{uw}, c_{uv} + c_{vw})$. Sinon, un arête (u, w) de coût $c_{uv}+c_{vw}$ est ajoutée. Ces contrôles peuvent être appliqués de manière itérative jusqu'à ce que le graphique reste inchangée. Dans la pratique, nous nous limitons à trois itérations. D'autres réductions ont été prises en considération, mais ils n'ont eu qu'un impact très marginal.

4 Expériences

Toutes les 960 instances de 60 nœuds présentées dans [6] sont reprises dans une classe **C1**. La classe **C2** contient 2500 instances générées à partir de 100 graphes non dirigés de 160 nœuds et 25 couples $\langle q, |S| \rangle$, variant de $\{\langle 3, 20\rangle$ à $\langle 119140\rangle\}$.

4.1 Comparaison

Nous comparons d'abord les approches MIP avec l'approche CP de [6]. La figure 1 donne un résumé des résultats expérimentaux. La colonne %*opt* donne le pourcentage d'instance résolues à l'optimalité dans le délai de 30 minutes, $\overline{I}$ le nombre moyen d'itérations, $\overline{N}$ la moyenne du nombre de nœuds dans l'arbre de branch-and-bound, $\overline{C}$ la moyenne du nombre de contraintes séparées et $\overline{T}$ le temps de calcul moyen en secondes (sur les instances résolues). La figure 2 montre l'évolution du pourcentage d'instances résolues dans **C2** par rapport à la limite de temps.

Clairement, toutes les approches MIP surpassent l'approche CP. Dans les figures 1 et 2, il n'y a pas de grande différence entre *Mod3_B&C_lazy* et *Mod4_B&B*, ni entre *Mod1 _B&C_lazy* et *Mod3_B&C_dyn* . *Mod3_B&C_lazy* et *Mod4 _B & B* sont les deux meilleures approches. L'approche *Mod2_B&B* est la moins bonne des approches MIP, bien qu'il crée peu de nœuds. Le petit nombre de nœuds résulte du fait que la version de relaxation entière de ce modèle est assez proche de la solution optimale. Notons que *Mod3_B&C_dyn* a le plus petit nombre d'itérations. Cela vient principalement de l'approche de séparation de contraintes dynamique, qui produit des arbres branch-and-bound plus petits. Cependant, le nombre de contraintes supplémentaires est beaucoup plus grand.

Les résultats expérimentaux ont également montré que la propriété (2) des solutions de QRP (Section 2) est très utile pour tous les modèles proposés dans le présent papier. Par exemple, cette propriété permet de résoudre 6.2% plus d'instances dans la classe **C2**.

4.2 Effet des valeurs de q et $|S|$ sur la performance des approches

Nous analysons la sensibilité de la performance par rapport à la valeur de q et la taille de l'ensemble de nœuds de multidiffusion. Nous avons divisé les instances de la classe **C2** en deux ensembles de groupes. Dans le premier ensemble, chaque groupe contient toutes les instances ayant un ensemble de nœuds de multidiffusion de même taille. Ainsi, le groupe *G20* (resp. *G50, G80, G110* et *G140*) est constitué de toutes les instances avec $|S| = 20$ (resp. $50, 80, 110$ et 140). Dans le deuxième ensemble, un groupe est composé des 500 instances avec le même rapport $\frac{q}{|S|}$. Les groupes *G1* à *G5* se répartissent les instances de la plus petite à la plus grande valeur de $\frac{q}{|S|}$.

Les résultats expérimentaux pour chaque groupe sont repris figure 3. Les différentes approches ont dif-

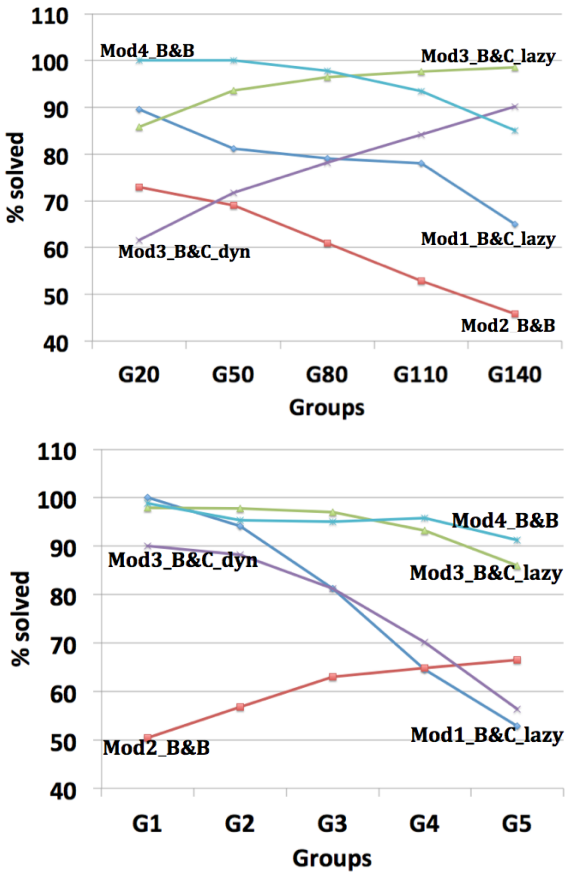


FIGURE 3 – Comparer les approches de MIP en résolvant des groupes d'instances en regardant le pourcentage d'instances résolues

férentes sensibilités de q. Les instances du groupe *G5* sont plus difficiles à résoudre que celles des autres groupes, sauf pour *Mod2_B&B*. Lorsque l'on considère le rapport $\frac{q}{|S|}$, *Mod3_B&_lazy* et *Mod3_B&_dyn* sont meilleures pour une grande valeur de ce rapport, tandis que les autres approches sont moins bonnes. Ces résultats confirment également que *Mod3_B&C_lazy* et *Mod4_B&B* sont les deux meilleures approches. Cependant, il existe une différence significative entre ces deux approches lorsque le nombre de nœuds de multidiffusion varie.

5 Conclusion

Ce papier propose une méthode de résolution exacte du problème de routage quorumcast basée sur la programmation entière. Nous avons proposé quatre formulations mathématiques pour QRP. Nous avons ensuite résolu QRP à l'optimalité en programmation entière, en utilisant notamment deux relaxations de contraintes. Les résultats expérimentaux ont montré que les approches MIP sont beaucoup plus efficaces que l'approche CP de l'état de l'art. En outre, nous avons montré que les deux approches, *Mod3_B&C_lazy* et *Mod4_B&B* sont les meilleures. Enfin, les résultats expérimentaux ont souligné que les différentes approches ont des sensibilités différentes aux paramètres q et la taille de l'ensemble de nœuds de multidiffusion. Comme piste de recherche future, de nouvelles séparations de contraintes pourraient être étudiées.

Références

[1] Michael Drexl and Stefan Irnich. Solving elementary shortest-path problems as mixed-integer programs. *OR Spectrum*, pages 1–16, 2012.

[2] Shun Yan Cheung and A. Kumar. Efficient quorumcast routing algorithms. In *INFOCOM '94. Networking for Global Communications., 13th Proceedings IEEE*, pages 840 –847 vol.2, jun 1994.

[3] Quoc Trung BUI, Quang Dung PHAM, and Yves DEVILLE Solving the Quorumcast Routing Problem as a Mixed Integer Program. In 11th International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR 2014), Cork, Ireland, May 2014, Lecture Notes in Computer Science, Springer, 2014.

[4] B. Du, J. Gu, D.H.K. Tsang, and W. Wang. Quorumcast routing by multispace search. In *Global Telecommunications Conference, 1996. GLOBECOM '96. 'Communications : The Key to Global Prosperity*, volume 2, pages 1069 –1073 vol.2, nov 1996.

[5] Chor Ping Low. A fast search algorithm for the quorumcast routing problem. *Inf. Process. Lett.*, 66(2) :87–92, April 1998.

[6] QuangDung Pham and Yves Deville. Solving the quorumcast routing problem by constraint programming. *Constraints*, 17 :409–431, 2012.

[7] B. Wang and J. C. Hou. An efficient QoS routing algorithm for quorumcast communication. *Computer Networks Journal*, 44(1) :43–61, 2004.

Ressources à décomposition continue et activités à durée dépendante d'intervalle pour l'ordonnancement de patients en médecine nucléaire (Extended Abstract)

Cyrille Dejemeppe **Yves Deville**

ICTEAM, Université Catholique de Louvain (UCLouvain), Belgique
{cyrille.dejemeppe, yves.deville}@uclouvain.be

Résumé

Cet article est un résumé de [2]. La médecine nucléaire est une technique d'imagerie médicale dans laquelle des traceurs nucléaires sont injectés aux patients. L'émission radioactive de la décomposition de traceurs dans le corps des patients est capturée pour générer une image de diagnostic. L'ordonnancement journalier des patients dans un centre de médecine nucléaire est un problème compliqué et intéressant à cause de la décomposition continue des traceurs radioactifs. Nous définissons deux nouveaux concepts d'ordonnancement pour résoudre ce problème : les ressources à décomposition continue et les activités à durée dépendante d'intervalle. Dans cet article, nous modélisons ce problème comme un COP ; nous proposons une stratégie de résolution utilisant la programmation par contraintes avec du LNS, et nous développons le processus de propagation pour traiter les deux nouvelles abstractions d'ordonnancement introduites.

1 Introduction

La médecine nucléaire est une pratique clinique dans laquelle des traceurs nucléaires sont injectés aux patients pour obtenir des images de diagnostic d'un large panel de maladies. Un traceur nucléaire est un ensemble de composants radioactifs dont la radioactivité décroît en fonction du temps selon la loi de décomposition définie dans [3] :

$$Rad(t) = Rad_0 \times \mathrm{e}^{\frac{-t\ln(2)}{t_{0.5}}} \quad (1)$$

où Rad_0 est la radioactivité initiale du composant radioactif et $t_{0.5}$ est le temps de demi-vie de celui-ci. Durant sa décomposition, les composants radioactifs d'un traceur émettent des rayons gamma ou des photons à haute énergie. Des senseurs externes capturent ces émissions et une image est construite par ordinateur. Comme expliqué dans [1], la médecine nucléaire a de nombreux avantages par rapport aux autres techniques d'imagerie médicale.

Le problème d'ordonnancement des patients dans un centre de médecine nucléaire (NMP) consiste en l'optimisation de l'ordonnancement du traitement des patients dans un centre de médecine nucléaire. Les patients d'un centre de médecine nucléaire sont traités selon une séquence d'étapes. En premier lieu, un traceur nucléaire est injecté au patient. Ensuite, il doit attendre que son corps incorpore le traceur. Ce temps d'attente a une durée minimale et maximale de sorte à ce que le corps n'émette ni trop ni trop peu de photon lors de l'acquisition de l'image. Enfin, le patient se rend dans une salle de scanner où l'acquisition de l'image a lieu. Comme expliqué dans [7], le temps que dure l'acquisition d'image pour un patient peut être exprimé comme une fonction linéaire du temps d'attente du patient après son injection de traceur :

$$\text{temps acquisition} = \alpha + \delta \times \text{temps attente} \quad (2)$$

où α et δ sont des constantes positives qui dépendent de la quantité et du type de traceur injecté.

La quantité de traceur injectée à un patient doit correspondre à un certain niveau de radioactivité qui dépend du type d'examen désiré. Comme la radioactivité d'un traceur décroît en fonction du temps, il est nécessaire d'injecter un volume de traceur qui augmente en fonction du temps pour qu'une quantité définie de traceur lui soit injecté.

NMP est un problème d'optimisation qui considère deux objectifs disjoints. Le premier objectif est de traiter les patients de la journée sur une période de temps la plus courte possible. Le seconde objectif est de limiter l'utilisation totale de traceur radioactif sur une journée.

Nous modélisons NMP comme un problème d'ordonnancement. Pour modéliser la décomposition des traceurs nucléaires, nous introduisons deux nouvelles abstractions d'ordonnancement : les ressources à décomposition continue et les activités à durée dépendante d'intervalle.

Cet article a aussi pour vocation d'illustrer comment résoudre les problèmes d'ordonnancement cumulatif avec des contraintes additionnelles non-triviales. Dans ce but, le modèle et la recherche pour résoudre NMP seront exprimées de manière aussi générique que possible.

2 Le modèle

Nous modélisons NMP comme un problème d'ordonnancement qui peut donc être caractérisé par ses quatre composants principaux : activités, ressources, contraintes et objectifs.

Les activités du modèle sont les étapes par lequel un patient doit passer durant son traitement. Les activités d'un patient peuvent être groupées dans un job ordonné j contenant deux activités : injection et acquisition d'image. Les ressources sont les injecteurs et les salles de scanner, qui sont modélisées comme des ressources cumulatives, ainsi que les traceurs radioactifs qui sont modélisés à l'aide d'une nouvelle abstraction : les ressources à décomposition continue. Le modèle comporte des contraintes de précédence pour faire en sorte que les activités au sein d'un même job restent ordonnées. Les ressources impliquent aussi l'ajout de contraintes pour faire en sorte que leur capacité ne soit jamais dépassée. Il faut aussi ajouter des contraintes pour lier la durée du temps d'acquisition d'image à la durée du temps d'attente du patient. Enfin, il faut ajouter des contraintes qui bornent le temps d'attente d'un patient, i.e. qui imposent un délai minimal et un délai maximal entre une injection et une acquisition d'image. L'objectif qui consiste à traiter les patients le plus rapidement possible est modélisé par une minimisation du makespan. Le second objectif de minimisation de la consommation totale de traceur est modélisé par une minimisation de la somme de la consommation de traceur par les patients. Nous exprimons notre modèle d'ordonnancement plus formellement comme un COP $\langle X, D, C, O\rangle$ comme suit.

X **: Les variables** Les variables de notre problème sont les temps de début et de fin de chaque activité. Soit une instance contenant k jobs qui eux-mêmes contiennent n activités, pour chaque activité $A_{i,j}$ ($j^{ème}$ activité du $i^{ème}$ job), nous définissons les variables début$(A_{i,j})$ et fin$(A_{i,j})$ avec $1 \leq i \leq k$ et $1 \leq j \leq n$.

D **: Les domaines** Les variables début$(A_{i,j})$ ont pour domaine $[0; \text{horizon} - \text{durée}(A_{i,j})]$ (avec $1 \leq i \leq k$, $1 \leq j \leq n$) si leur durée est fixée et $[0; \text{horizon}]$ sinon. Similairement, les variables fin$(A_{i,j})$ ont pour domaine $[\text{durée}(A_{i,j}); \text{horizon}]$ si leur durée est fixée et $[0; \text{horizon}]$ sinon.

C **: Les contraintes** Premièrement nous imposons des contraintes de précédence qui ordonnent les jobs et fixent un délai minimum entre deux activités successives quand c'est nécessaire : $\text{fin}(A_{i,j}) + \text{délai}_{min}(A_{i,j}, A_{i,j+1}) \leq \text{début}(A_{i,j+1})$ avec $1 \leq i \leq k$, $1 \leq j \leq n-1$ et $\text{délai}_{min}(A_{i,j}, A_{i,j+1})$ le délai minimal entre les deux activités qui est nul quand aucun délai n'est requis entre les activités considérées. En plus de ce délai minimal, un délai maximal doit être imposé entre l'injection et l'acquisition d'image pour un patient : $\text{début}(A_{i,j+1}) - \text{fin}(A_{i,j}) \leq \text{délai}_{max}(A_{i,j}, A_{i,j+1})$ avec $\text{délai}_{max}(A_{i,j}, A_{i,j+1})$ le délai maximum entre les deux activités.

Nous devons ajouter d'autres contraintes pour lier les variables de début et de fin d'une activité à la durée de celle-ci : $\text{fin}(A_{i,j}) - \text{début}(A_{i,j}) = \text{durée}(A_{i,j})$. Lorsque la durée d'une activité dépend de l'intervalle entre l'activité qui la précède et elle-même, nous devons lier ces variables par une fonction f (monotone croissante). C'est ce qu'on appelle les *activités à durée dépendante d'intervalle* : $\text{fin}(A_{i,j}) - \text{début}(A_{i,j}) = f(\text{début}(A_{i,j}) - \text{fin}(A_{i,j-1}))$.

Notre modèle contient aussi des contraintes liées aux ressources cumulatives qui imposent que leur capacité ne soit jamais dépassée : $\sum_A \text{utilisation}(A, R_{cum}, t) \leq \text{capacité}(R_{cum})$ où $0 \leq t \leq horizon$ et $\text{utilisation}(A, R_{cum}, t)$ est l'utilisation de la ressource R par l'activité A en t.

Enfin, nous devons ajouter la contrainte sur les ressources à décomposition continue. Nous modélisons cette contrainte en augmentant la quantité de ressource requise par une activité en fonction du temps auquel cette activité débute. La quantité de ressource à décomposition continue R_{dec} nécessaire à une activité $A_{i,j}$ qui commence au temps t est exprimée comme suit :

$$q(A_{i,j}, R_{dec}, t) = \frac{C_i}{Rad(t)} = \frac{C_i}{Rad_0} \times e^{\frac{t \ln(2)}{t_{0.5}}} \qquad (3)$$

où $Rad(t)$ est défini dans 1, Rad_0 et $t_{0.5}$ dépendent de R_{dec}, et C_i est une constante positive dépendant de la radioactivité dont le patient i a besoin. Cette formule exprime le fait que la quantité de traceur injectée à un patient est inversément proportionnelle à la radioactivité du traceur. Nous devons ajouter une contrainte pour imposer que la consommation totale de traceur est inférieure ou égale à la capacité initiale de la ressource :

$$\sum_A q(A, R_{dec}, \text{début}(A)) \leq \text{capacité initiale}(R_{dec}) \qquad (4)$$

O **: Les objectifs** Le premier objectif est de minimiser le makespan : minimiser $makespan = \max_A(\text{fin}(A))$. Le second objectif est de minimiser la consommation totale de ressource à décomposition continue : minimiser $\sum_{R_{dec}} \sum_A w_{dec} \times q(A, R_{dec}, \text{début}(A))$ où w_{dec} sont des poids positifs.

3 Propagation

Dans cette section, nous expliquons les processus de propagation associés aux ressources à décomposition continue et aux activités à durée dépendante d'intervalle.

3.1 Ressources à décomposition continue

Le processus de propagation que nous décrivons permet d'obtenir la cohérence de borne pour la contrainte explicitée dans l'équation 4. Soit une activité $A_{i,j}$ et une ressource à décomposition continue R_{dec} donnée, la quantité de ressource nécessaire par l'activité est une fonction croissante monotone en fonction du temps t : $q(A_{i,j}, R_{dec}, t)$. Étant donné que la quantité de traceur à injecter à un patient est inversément proportionnelle à la radioactivité du traceur, grâce à l'équation 1, nous pouvons écrire :

$$q(A_{i,j}, R_{dec}, t) = \beta_{i,j} \times \mathrm{e}^{\frac{t \ln(2)}{\gamma_{dec}}} \tag{5}$$

où $\beta_{i,j}$ est une constante positive qui dépend de $A_{i,j}$ et γ_{dec} est une constante qui dépend de R_{dec}. Nous pouvons donc réécrire la contrainte décrite dans l'équation 4 comme suit :

$$\sum \beta_{i,j} \times \mathrm{e}^{\frac{\text{début}(A_{i,j}) \ln(2)}{\gamma_{dec}}} \leq \text{capacité initiale}(R_{dec}) \tag{6}$$

Le concept de *dérivation de propagateur basé sur des vues* introduit dans [6] nous permet de réécrire cette contrainte comme une contrainte de somme linéaire. Étant donné un propagateur p, une vue est représentée par deux fonctions ϕ et ϕ^{-1} qui sont composées avec p pour obtenir le propagateur désiré $\phi \circ p \circ \phi^{-1}$. La fonction ϕ transforme le domaine d'entrée et ϕ^{-1} applique la transformation inverse au domaine de sortie renvoyé par le propagateur. Pour pouvoir appliquer ce mécanisme à notre contrainte décrite dans l'équation 6, nous définissons une fonction $\phi_{i,j}$ pour chaque variable début$(A_{i,j})$ présente dans cette contrainte :

$$\phi_{i,j}(v) = \beta_{i,j} \times \mathrm{e}^{\frac{v \ln(2)}{\gamma_{dec}}} \tag{7}$$

La fonction inverse $\phi_{i,j}^{-1}$ est définie comme suit :

$$\phi_{i,j}^{-1}(v) = \frac{\gamma_{dec}}{\ln(2)} \times \ln\left(\frac{v}{\beta_{i,j}}\right) \tag{8}$$

Grâce à ces définition de ϕ et ϕ^{-1}, nous pouvons utiliser un algorithme classique de propagation sur une contrainte de somme linéaire pour propager les contraintes de ressources à décomposition continue.

3.2 Activités à durée dépendante d'intervalle

Le processus de propagation que nous proposons permet d'obtenir la cohérence de cohérence pour la contrainte sur les activités à durée dépendante d'intervalle. Comme explicité précédemment, la fonction f liant la durée d'une activité à un intervalle est une fonction monotone, linéaire et croissante. De cette affirmation, nous pouvons réécrire la contrainte sur les activités à durée dépendante d'intervalle comme suit :

$$\begin{aligned} \text{fin}(A_{i,j}) - \text{début}(A_{i,j}) = \varepsilon_{i,j} + \delta_{i,j} \times \\ \big(\text{début}(A_{i,j}) - \text{fin}(A_{i,j-1})\big) \end{aligned} \tag{9}$$

où $\delta_{i,j}$ et $\varepsilon_{i,j}$ sont des constantes positives. Utilisant une approche similaire à celle décrite dans la section 3.1, il est possible d'utiliser des vues triviales pour propager cette contrainte en utilisant un algorithme de propagation classique de contrainte d'égalité de somme linéaire. Dès lors, il est possible de propager les contraintes sur les activités à durée dépendante d'intervalle.

4 Résultats expérimentaux

Pour donner une vision globale de la nature complexe du problème traité, nous résolvons quatre versions différentes du problème. Ces versions sont résolues à l'aide de la programmation par contraintes avec du LNS, introduit dans [8], et l'heuristique de branchement est un branchement binaire du premier échec. Une limite de trois minutes est imposée pour la résolution de ces versions et les meilleurs valeurs obtenues pour les deux objectifs (makespan et quantité de traceur consommé) sont rapportées dans la table 1. Toutes les expériences ont été réalisées avec le solveur open-source OscaR [4]. Les instances considérées ont été générées par une générateur aléatoire avec biais (réaliste) que nous avons conçu. Les centres de médecine nucléaire traitent en général 25 patients par jour et les instances de taille plus importante sont considérées pour tester les limites du modèle.

La première version V_1 est une relaxation du problème qui ne considère ni les ressources à décomposition continue, ni les activités à durée dépendante d'intervalle et qui minimise uniquement le makespan. Nous pouvons observer dans la table 1 que la quantité de traceur (QT) croît grandement avec le nombre de patient et le makespan, ce qui est dû à la nature exponentielle de la quantité de traceur requise en fonction du temps.

La version V_2 considère les activités à durée dépendante d'intervalle et minimise uniquement le makespan. Nous pouvons voir dans la table 1 que le makespan obtenu pour V_2 est plus grand que pour V_1. Cela s'explique par deux raisons. Premièrement, les solutions de V_1 ne sont pas des solutions pour V_2 (à cause de l'ajout de la contrainte sur les activités à durée dépendante d'intervalle). Deuxièmement, comme V_2 relâche les durées des activités, l'espace de recherche est plus grand et il faut plus de temps pour converger vers une solution de qualité que dans V_1.

La version V_3 ajoute à V_2 les ressources à décomposition continue et minimise uniquement la quantité totale de traceur utilisé. Nous observons dans la table 1 que la quantité totale de traceur utilisé (QT) est en moyenne inférieure pour V_3 que pour V_2 alors que c'est l'inverse pour le makespan.

La version V_4 est la version bi-objectif de V_3 et minimise le makespan et la quantité totale de traceur. Cette version va donc obtenir un ensemble de solutions non-dominées au sens de Pareto en utilisant une variante de la contrainte introduite dans [5]. Nous pouvons voir dans la table 1 que

les valeurs reportées se trouvent entre les meilleurs et les pires résultats obtenus par V_2 et V_3 pour les deux objectifs. Dans la figure 1, nous pouvons observer un exemple de front de Pareto obtenu sur une instance de 20 patients.

Version du problème	20 Patients		40 Patients		50 Patients	
	MS	QT	MS	QT	MS	QT
Problème V_1	446	40.44	867	516	1,048	1,268
Problème V_2	486	51.16	994	1,065	1,211	2,770
Problème V_3	530	39.04	1,029	671	1,245	1,862
Problème V_4	*495*	*38.32*	*1,011*	*757*	*1,234*	*1,885*

TABLE 1 – Valeurs moyennes des objectifs pour différentes versions du problème et différentes tailles d'instance. MS représente le makespan et QT est la quantité de traceur. Pour les versions V_1, V_2 et V_3, les valeurs rapportées sont les valeurs moyennes pour la taille d'instance considérée. Pour la version V_4, les valeurs rapportées sont les moyennes des meilleures valeurs trouvées pour chaque objectif pour la taille considérée.

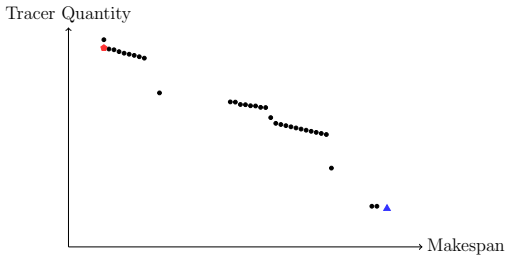


FIGURE 1 – Comparaison du front de Pareto obtenu par V_4 et des meilleures solutions obtenues par V_2 et V_3 sur une instance de 20 patients. Le carré rouge est la meilleure solution de V_2, le triangle bleu est la meilleure solution obtenue par V_3 et les cercles noirs sont les solutions du front de Pareto obtenu par V_4.

5 Conclusion

Dans cet article, nous avons décrit le problème d'ordonnancement des patients dans un centre de médecine nucléaire et nous l'avons modélisé comme un problème d'ordonnancement. Pour ce faire, nous avons introduit deux nouvelles abstractions : les ressources à décomposition continue et les activités à durée dépendante d'intervalle. Ces abstractions ont été modélisées à l'aide de vues et de contraintes linéaires de somme. Enfin, nous avons proposé une méthode efficace pour résoudre ce problème avec la programmation par contraintes et le LNS. Nous avons été capables de résoudre ce problème et d'obtenir une approximation correct du front de Pareto des solutions en un temps raisonnable.

Le modèle et la recherche décrite dans cet article sont génériques et peuvent être appliqués à un grand nombre d'autres problèmes cumulatifs d'ordonnancement avec contraintes additionnelles non-triviales. Grâce à l'utilisation de vues et de propagateurs existants, ces contraintes additionnelles peuvent être ajoutées à un modèle standard d'ordonnancement cumulatif.

Acknowledgments Les auteurs souhaitent remercier les relecteurs anonymes pour leurs commentaires avisés. Cette recherche est subventionnée par le projet Mirror, le projet FRFC 2.4504.10 du FNRS de Belgique et l'action de recherche concertée ICTM22C1 de l'UCLouvain.

Références

[1] S.R. Cherry, J.A. Sorenson, and M.E. Phelps. *Physics in Nuclear Medicine*. Elsevier Health Sciences, 2012.

[2] Cyrille Dejemeppe, Yves Deville, et al. Continuously degrading resource and interval dependent activity durations in nuclear medicine patient scheduling. In *11th International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR 2014)*, 2014.

[3] Joanna S. Fowler and Tatsuo Ido. Initial and subsequent approach for the synthesis of 18fdg. *Seminars in Nuclear Medicine*, 32(1) :6 – 12, 2002. Impact of FDG-PET Imaging on the Practice of Medicine.

[4] OscaR Team. OscaR : Scala in OR, 2012. Available from `https://bitbucket.org/oscarlib/oscar`.

[5] Pierre Schaus and Renaud Hartert. Multi-objective large neighborhood search. In *Principles and Practice of Constraint Programming*, pages 611–627. Springer Berlin Heidelberg, 2013.

[6] Christian Schulte and Peter J. Stuckey. Efficient constraint propagation engines. *Transactions on Programming Languages and Systems*, 31(1) :2 :1–2 :43, December 2008.

[7] Heiko Schöder, YusufE. Erdi, StevenM. Larson, and HenryW.D. Yeung. Pet/ct : a new imaging technology in nuclear medicine. *European Journal of Nuclear Medicine and Molecular Imaging*, 30(10) :1419–1437, 2003.

[8] Paul Shaw. Using constraint programming and local search methods to solve vehicle routing problems. In *Principles and Practice of Constraint Programming—CP98*, pages 417–431. Springer, 1998.

La domaine k-intercohérence aussi simplement que la cohérence d'arc
(Résumé Étendu)

Jean-Baptiste Mairy[1] **Yves Deville**[1] **Christophe Lecoutre**[2]

[1] ICTEAM, Université catholique de Louvain, Belgique
[2] CRIL-CNRS UMR 8188, Université d'Artois, F-62307 Lens, France
{jean-baptiste.mairy,yves.deville}@uclouvain.be lecoutre@cril.fr

Résumé

Ce papier est un résumé étendu de [11]. Il définit une cohérence combinant la cohérence d'arc généralisée (GAC) avec la k-intercohérence (kWC) : la domaine k-intercohérence (DkWC). Cette cohérence est plus forte que GAC et correspond au filtrage au niveau des domaines qu'obtiendrait GAC sur un CSP qui est déjà kWC. Ensuite, il introduit une procédure pour appliquer DkWC, basée sur un CSP modifié, qui permet d'obtenir la DkWC uniquement avec des propagateurs GAC existants. Finalement, il montre les bénéfices de cette approche simple et facilement intégrable dans un solveur existant par des expériences sur une variété de problèmes.

1 Introduction

La propagation, caractérisée par des propriétés dites de cohérence, est un concept clé en programmation par contraintes (CP). La cohérence centrale en CP est la cohérence d'arc généralisée (GAC) [10]. Il existe une pléthore de cohérences plus faibles (réduisant moins l'espace de recherche) et plus fortes (réduisant plus l'espace de recherche) que GAC.

Les cohérences réduisant les domaines des variables, comme GAC, peuvent être cataloguées comme 'basées sur les domaines' (voir [1, 3, 2, 6, 8, 13]). Une autre catégorie existe : les cohérences 'basées sur les contraintes'. Les cohérences de cette dernière catégorie identifient des tuples de valeurs ne pouvant mener à une solution. Ces cohérences modifient donc les contraintes et leur intégration dans un solveur peut être compliquée. La combinaison d'une cohérence basée sur les contraintes avec une cohérence basée sur les domaines permet une meilleure réduction du domaine des variables. C'est ce que nous proposons dans ce papier : une combinaison de GAC avec une cohérence basée sur les contraintes : la k-intercohérence (kWC). Il est à noter que ces deux cohérences ont déjà été théoriquement combinées dans [5, 4]. Elles ont également été combinées en pratique dans [2, 13] mais sous une forme affaiblie. Un algorithme sophistiqué est également proposé dans [6] pour la combinaison complète.

Dans ce papier, nous définissons tout d'abord la domaine k-intercohérence (DkWC). Cette cohérence, basée sur les domaines, est plus forte que GAC. Ensuite, nous définissons une procédure simple et pratique pour appliquer DkWC, basée sur un CSP modifié, qui permet d'obtenir la DkWC uniquement avec des propagateurs GAC existants. Finalement, nous montrons les bénéfices de cette approche simple et facilement intégrable dans un solveur existant par des expériences sur une variété de problèmes.

2 Notations

Un problème de satisfaction de contraintes $P = (X, D, C)$ est composé d'un ensemble ordonné de n variables $X = \{x_1, \ldots, x_n\}$, un ensemble de domaines $D = \{D(x_1), \ldots, D(x_n)\}$ où $D(x_i)$ contient l'ensemble fini des valeurs possibles pour x_i et un ensemble de e contraintes $C = \{c_1, \ldots, c_e\}$ où chaque contrainte c_j restreint les combinaisons possibles des valeurs, appelées tuples permis, d'un sous-ensemble ordonné de variables de X ; ce sous-ensemble est appelé portée de c_j et sera noté $scp(c_j)$. $D(x)$ fait référence au domaine courant pour x, qui est un sous-ensemble du domaine initial de x, noté $D^{init}(x)$. Pour un sous-ensemble Y de X, $D[Y]$ est la restriction de D aux variables de Y.

Dans ce papier, nous considérons des contraintes table (positives), qui sont des contraintes données en

extension par la liste $rel(c)$ des tuples permis. Nous utiliserons τ pour désigner un tuple. Une contrainte table est respectée ssi $(x_1, \ldots, x_r) \in rel(c)$, où $scp(c) = \{x_1, \ldots, x_r\}$. Si $\tau = (a_1, \ldots, a_r)$ est un tuple de longueur r, alors, $\tau \odot b$ représente le tuple $(a_1, \ldots, a_r, b)$ de longueur $r + 1$.

Une contrainte c est satisfaite par un tuple $\tau \in D^{init}(scp(c))$ ssi τ est un tuple permis par c. Nous utiliserons $c(\tau)$, ou, de façon équivalente $\tau \in c$, pour désigner le test de satisfaction de c par τ. Pour un tuple τ, associé implicitement à un ensemble ordonné de variables comprenant x, $\tau[x]$ indique la valeur dans τ associée à la variable x. Dans ce papier, nous utiliserons la notion de tuple et d'assignation (une valeur affectée à chaque variable) de façon interchangeable. Un tuple ou une assignation τ sur $Y \subseteq X$ est dite valide si $\tau \in D(Y)$. Nous appellerons littéral un couple variable-valeur. Un littéral d'une contrainte c est un couple (x, v) où $x \in scp(c)$ et $a \in D(x)$.

3 Domaine k-intercohérence

La cohérence définie dans ce papier est une combinaison de la cohérence d'arc généralisée (GAC), aussi appelée parfois cohérence de domaine (DC), avec la k-intercohérence (kWC).

La définition de GAC fait appel à la notion de support pour un littéral. Un support pour un littéral (x, v) d'une contrainte c est un tuple $\tau \in D(scp(c))$ tel que $\tau \in c$ et $\tau[x] = v$.

Définition 1 *(GAC) Une contrainte c est arc-cohérente (GAC) ssi il existe au moins un support pour chaque littéral de c. Un CSP est GAC ssi toutes ses contraintes sont GAC.*

GAC est une cohérence basée sur les domaines et son application revient à retirer du domaine des variables les valeurs qui n'ont pas de support. La k-intercohérence (kWC) [5, 1], quant à elle, est basée sur les contraintes. Son application revient à retirer des contraintes les tuples qui ne peuvent pas être étendus.

Définition 2 *(Extension) Soient Y et Z, deux ensembles de variables. Une assignation τ' de $Y \cup Z$ est une extension sur $Y \cup Z$ d'une assignation τ de Y ssi $\tau'[y] = \tau[y], \forall y \in Y$. Une extension valide est une extension qui correspond à une assignation valide.*

Définition 3 *(*kWC*) Un CSP $P = (X, D, C)$ est k-intercohérent (kWC) ssi $\forall c_1 \in C$, $\forall \tau \in c_1 : \tau \in D(scp(c_1))$, $\forall c_2, \ldots, c_k \in C, \exists \tau'$ extension valide de τ sur $\bigcup_{i=1}^{k} scp(c_k)$ satisfaisant $c_2, \ldots, c_k$.*

Les cohérences kWC et GAC sont incomparables [4], ce qui veut dire qu'un CSP peut être kWC mais pas GAC et inversement. Cela signifie aussi que l'application conjointe des deux cohérences (GAC+kWC) permet un filtrage plus fort que les deux cohérences considérées de manière isolée. De façon intéressante, une cohérence basée sur les domaines peut être définie à partir de GAC+kWC. Nous appelons cette cohérence la domaine k-intercohérence (DkWC) et elle correspond au filtrage de GAC+kWC au niveau du domaine des variables uniquement.

Définition 4 *(DkWC) Un CSP $P = (X, D, C)$ est domaine k-intercohérent (DkWC) ssi GAC+kWC(P) est un CSP $Q = (X, D^Q, C^Q)$ tel que $D = D^Q$.*

4 Procédure de filtrage pour la DkWC

La procédure de filtrage pour DkWC se base sur une généralisation à la kWC du filtrage proposé dans [4], et différente de celle présentée dans [5]. Grâce à leur accès explicite aux tuples permis, les contraintes table sont particulièrement adaptées aux cohérences basées sur les contraintes (comme la kWC). Nous ne considérons dès lors plus que des contraintes table. L'idée du filtrage pour la kWC est de définir une forme duale du CSP à filtrer, appelé le k-dual. Le k-dual possède une variable x'_i par contrainte c_i du CSP d'origine, son domaine représentant (à l'aide d'un indice) l'ensemble des tuples de la contrainte qu'elle représente. Le k-dual possède une contrainte par groupe de k contraintes originales, représentant leur jointure.

Définition 5 *(CSP k-dual) Soit $P = (X, D, C)$, un CSP. Le k-dual de P est un CSP $P^{kd} = (X^{kd}, D^{kd}, C^{kd})$, où :*

- *pour chaque contrainte $c_i \in C$, X^{kd} contient une variable x'_i avec, pour domaine $D^{kd}(x'_i) = \{1, 2, \ldots, |rel(c_i)|\}$,*
- *pour chaque sous ensemble S de k contraintes de C, C^{kd} contient une contrainte c' telle que $scp(c') = \{x'_i \mid c_i \in S\}$ et c' est une contrainte d'arité k représentant la jointure des contraintes de S (en utilisant les indices de tuples).*

Pour le k-dual d'un CSP, nous avons la propriété suivante :

Propriété 1 *Un CSP est kWC ssi son k-dual est GAC [4, 5].*

Pour obtenir la domaine k-intercohérence (DkWC) avec uniquement des propagateurs GAC existants, l'idée est de définir un CSP contenant conjointement le CSP original et son k-dual. Bien sûr, pour que la suppression des tuples des contraintes (dans les variables k-duales) se reflète au niveau du domaines des variables originales et vice-versa, un lien est ajouté entre les contraintes originales et les variables k-duales. Ce lien est obtenu en transformant chaque contrainte originale c_i en une contrainte hybride $\phi(c_i)$. La contrainte hybride $\phi(c_i)$ a pour portée celle de la contrainte originale c_i, avec en plus la variable k-duale x'_i correspondant à c_i. Les tuples de $\phi(c_i)$ sont les tuples de c_i avec en plus leur indice dans $rel(c_i)$.

Définition 6 *(Contraintes Hybrides) Soit $P = (X, D, C)$ un CSP. L'ensemble de contraintes hybrides $\phi(C)$ de P est l'ensemble $\{\phi(c_i) \mid c_i \in C\}$ où :*
- *$scp(\phi(c_i)) = scp(c_i) \cup \{x'_i\}$*
- *$rel(\phi(c_i)) = \{\tau_j \odot j \mid$ τ_j est le $j^{ème}$ tuple de $rel(c_i)\}$*

avec x'_i, la variable k-duale correspondant à c_i.

Nous pouvons maintenant définir le CSP qui sera utilisé pour obtenir la DkWC uniquement avec des propagateurs GAC : le CSP k-intercalé.

Définition 7 *(CSP k-intercalé) Soit $P = (X, D, C)$, un CSP. Le* k-intercalé *de P est le CSP $P^{ki} = (X^{ki}, D^{ki}, C^{ki}) = (X \cup X^{kd}, D \cup D^{kd}, \phi(C) \cup C^{kd})$ où (X^{kd}, D^{kd}, C^{kd}) est le k-dual de P, et $\phi(C)$ les contraintes hybrides de P.*

La propriété 2 montre une caractéristique importante des CSPs k-intercalés : appliquer un filtrage GAC sur un CSP k-intercalé atteint le même niveau de filtrage que GAC+kWC au niveau du domaine des variables.

Propriété 2 *Soient $P = (X, D, C)$ un CSP et $P^{ki} = (X^{ki}, D^{ki}, C^{ki})$ le k-intercalé de P. Si $Q = (X, D^Q, C^Q)$ est la fermeture GAC+kWC de P et $R = (X^{ki}, D^R, C^{ki})$ la fermeture GAC de P^{ki}, alors $D^Q = D^R[X]$ (i.e., $D^Q(x) = D^R(x), \forall x \in X$).*

De cette propriété découle le corollaire suivant.

Corollaire 1 *Si le k-intercalé d'un CSP P est GAC, alors P est DkWC.*

La complexité du filtrage DkWC est la complexité d'appliquer un filtrage GAC sur le CSP k-intercalé. Comme complexité pour GAC, nous utiliserons la complexité optimale du propagateur défini dans [12]. Soit P un CSP avec n variables, une taille maximale de domaine égale à d, e contraintes, un maximum de t tuples permis par contrainte et une arité maximale de r. Le k-intercalé de P est un CSP contenant $n' = n + e$ variables, une taille maximale de domaine égale à $d' = max(d, t)$, $e' = e + \binom{e}{k}$ [1] contraintes, une borne supérieure de $t' = t^k$ tuples par contrainte et une arité maximale de $r' = max(r+1, k)$. Appliquer un filtrage GAC sur le CSP k-intercalé avec le propagateur optimal pour contraintes table a donc une complexité en $O(e' \cdot (r' \cdot t' + r' \cdot d')) = O((\binom{e}{k} + e) \cdot (r' \cdot t' + r' \cdot d'))$.

Le prix à payer pour atteindre la DkWC est élevé pour deux raisons principalement : le nombre $\binom{e}{k}$ de contraintes k-duales à considérer, et leur taille qui est bornée par t^k. C'est pourquoi nous proposons deux variantes de notre procédure de filtrage. Ces deux variantes n'intègrent pas toutes les contraintes du k-dual. Le filtrage obtenu est toujours plus fort que GAC mais plus faible que DkWC. La première de ces deux variantes est appelée DkWC^{cy}. Dans cette variante, seuls les groupes de k contraintes originales formant un cycle dans le graphe de contraintes du problème original sont intégrés, sous forme de contraintes k-duales, dans le CSP k-intercalé. La seconde variante du filtrage DkWC va encore plus loin en ne considérant que les groupes de k contraintes formant un cycle dans le graphe de contraintes d'origine, dont la taille de la jointure est inférieure à un paramètre. Nous appellerons cette version DkWC^{cy-}.

1. $\binom{e}{k}$ est le nombre de sous-ensembles de taille k d'un ensemble de taille e

5 Résultats expérimentaux

Le tableau 1 est un résumé de nos résultats expérimentaux. Les résultats complets peut être trouvés dans [11]. Pour chacun de nos tests, la DkWC est maintenue durant la recherche. Cependant, comme expliqué en Section 4, inclure toutes les contraintes k-duales est trop coûteux pour beaucoup de problèmes en pratique. Nous comparons donc les versions faibles de notre procédure de filtrage : DkWC^{cy}et DkWC^{cy-}. Nous avons également porté notre attention sur la 3WDC et 4WDC. En effet, une valeur de k égale à 3 où 4 permet de réduire substantiellement l'espace de recherche par rapport à GAC tout en gardant le coût du filtrage raisonnable. Le branchement n'est réalisé que sur les variables du problème original et toutes les solutions sont cherchées. Le propagateur GAC existant utilisé pour filtrer le CSP k-intercalé est le propagateur optimal défini dans [12].

Dans le tableau 1, nous comparons notre filtrage avec le filtrage GAC obtenu par [12] (colonne *GAC*), le propagateur maxRPWC3 (colonne *maxRPWC3*) défini dans [2] et le propagateur eSTRw (colonne *eSTRw*) proposé dans [9]. La colonne *wDkWC* représente notre filtrage : D3WC^{cy}pour les instances Bin, Tern, AIM, Pret et Lang, D4WC^{cy}pour Dubois et D3WC^{cy-}pour TSP and ModRen. Les instances Dubois ne contiennent pas de cycles de longueur 3 et les deux dernières séries contiennent trop de grandes tables pour permettre l'utilisation de D3WC^{cy}. Tous les algorithmes sont (ré-)implémentés en Comet. Les séries *Bin* et *Tern* contiennent chacun 50 instances générées aléatoirement (générateur de [15]), avec respectivement des contraintes binaires et ternaires. *AIM*, *Pret* and *Dubois* sont des instances provenant de la Compétition de solveurs CSP [14]. Les séries Langford 2 (*Lang*), Travelling Salesman 20 (*TSP*) et Renault modifiées (*modRen*) viennent de [7]. Les tests ont été effectués sur un Intel Xeon 2.53GHz avec Comet 2.1.1. Nous avons utilisé un temps limite d'exécution de 20 minutes pour chaque instance. Le tableau donne

Bench	GAC		maxRPWC3		eSTRw		wDkWC	
	T	%sol	T	%sol	T	%sol	T	%sol
Bin	9.9	**100**	72	98	11.4	**100**	**2.9**	**100**
Tern	23.1	**100**	124	90	16.6	**100**	**12.6**	**100**
AIM	82	46	14.7	46	**1.2**	50	3.4	**88**
Pret	160	**50**	977	**50**	504	**50**	**132**	**50**
Lang	**0.5**	**58**	44.6	46	1.5	54	10.5	50
Dubois	793	15	-	8	598	15	**201**	**30**
TSP	**52**	**93**	-	33	233	80	94	**93**
ModRen	-	6	743	26	-	0	**502**	**34**

TABLE 1 – Résultats expérimentaux. T est le temps moyen de résolution en secondes et %sol est le pourcentage d'instances résolues.

le temps d'exécution moyen par instance en seconde. Pour la comparaison, nous ne pouvons utiliser que les instances résolues par toutes les techniques. Cependant, certaines techniques restreignent trop l'ensemble d'instances résolues. Dans la table, un '-' représente une technique qui a été exclue de la comparaison pour les raisons susmentionnées. Néanmoins, le pourcentage d'instances solutionnées est donné pour chaque technique (*%sol*).

A l'aide du tableau 1, nous pouvons remarquer que notre filtrage DkWC faible est plus rapide que maxRPWC3 sur toutes les séries. Il est également plus rapide que eSTRw sur toutes les séries hormis deux. Notre filtrage est aussi plus rapide que GAC sur toutes les séries exceptés deux, où GAC est plus rapide que toutes les techniques utilisant une cohérence forte. Notre filtrage est celui donnant lieu à la plus grande réduction de l'espace de recherche sur ces séries.

6 Conclusion

Cet article, résumé étendu de [11], présente une cohérence basée sur les domaines, DkWC, qui correspond au filtrage combiné de GAC et kWC au niveau des domaines. Nous avons défini une procédure de filtrage se basant sur un CSP modifié, appelé CSP k-intercalé, et sur des propagateurs GAC existants uniquement. Nous proposons également deux version faibles du filtrage. Des résultats expérimentaux montrent, sur un éventail de problèmes variés, l'efficacité de notre approche.

Remerciements Le premier auteur est financé comme Aspirant du FNRS belge. Cette recherche est aussi partiellement financée par le projet FRFC 2.4504.10 du FNRS belge, et l'Action de Recherche Concertée ICTM22C1 de l'ULCouvain. Le troisième auteur bénéficie d'un financement du CNRS et d'OSEO, au sein du projet ISI 'Pajero'.

Références

[1] C. Bessiere. Constraint propagation. In F. Rossi, P. van Beek, and T. Walsh, editors, *Handbook of Constraint Programming*. Elsevier, New York, USA, 2006.

[2] C. Bessiere, K. Stergiou, and T. Walsh. Domain filtering consistencies for non-binary constraints. *Artificial Intelligence*, 72(6-7) :800–822, 2008.

[3] R. Debruyne and C. Bessière. Domain filtering consistencies. *Journal of Artificial Intelligence Research*, 14 :205–230, 2001.

[4] P. Janssen, P. Jégou, B. Nouguier, and M.-C. Vilarem. A filtering process for general constraint-satisfaction problems : achieving pairwise-consistency using an associated binary representation. In *Proceedings of IEEE Workshop on Tools for Artificial Intelligence*, pages 420–427, 1989.

[5] P. Jégou. *Contribution à l'étude des Problèmes de Satisfaction de Contraintes : Algorithmes de propagation et de résolution. Propagation de contraintes dans les réseaux dynamique.* PhD thesis, Université de Montpellier II, 1991.

[6] S. Karakashian, R. Woodward, C. Reeson, B. Choueiry, and C. Bessiere. A first practical algorithm for high levels of relational consistency. In *Proceedings of AAAI'10*, pages 101–107, 2010.

[7] C. Lecoutre. *Instances of the Constraint Solver Competition.* http ://www.cril.fr/∼lecoutre/.

[8] C. Lecoutre. *Constraint Networks : Techniques and Algorithms.* ISTE/Wiley, 2009.

[9] C. Lecoutre, A. Paparrizou, and K. Stergiou. Extending STR to a higher-order consistency. In *Proceedings of AAAI'13*, pages 576–582, 2013.

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

[11] Jean-Baptiste Mairy, Yves Deville, Christophe Lecoutre, et al. Domain k-wise consistency made as simple as generalized arc consistency. In *11th International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR 2014)*, 2014.

[12] Jean-Baptiste Mairy, Pascal Van Hentenryck, and Yves Deville. Optimal and efficient filtering algorithms for table constraints. *Constraints*, 19(1) :77–120, 2014.

[13] K. Stergiou. Strong domain filtering consistencies for non-binary constraint satisfaction problems. *International Journal on Artificial Intelligence Tools*, 17(5) :781–802, 2008.

[14] M. van Dongen, C. Lecoutre, and O. Roussel. 2008 CSP solver competition. http ://www.cril.univ-artois.fr/CPAI08/.

[15] K. Xu, F. Boussemart, F. Hemery, and C. Lecoutre. Random constraint satisfaction : easy generation of hard (satisfiable) instances. *Artificial Intelligence*, 171(8-9) :514–534, 2007.

À propos de la bibliothèque de modèles XCSP

Arnaud Malapert[1] **Christophe Lecoutre**[2]

[1] Université Nice Sophia Antipolis, CNRS, I3S, UMR 7271, 06900 Sophia Antipolis, France
[2] Université d'artois, CNRS, CRIL, UMR 8188, rue de l'université, 62307 Lens cedex, France
arnaud.malapert@unice.fr lecoutre@cril.fr

Résumé

Les bibliothèques de modèles (modèles et instances) sont couramment utilisées pour évaluer les solveurs et les algorithmes en programmation par contraintes. Elles sont d'un intérêt certain pour l'expérimentation : vaste corpus de modèles, comparaison et reproductibilité. Toutefois, ces bibliothèques doivent être régulièrement examinées avec attention pour évaluer la pérennité de leur intérêt. Dans cet article, nous discutons de la classification des modèles, sous l'angle de la comparaison de solveurs. Ces réflexions sont étayées par les résultats d'un portfolio de solveurs utilisant les solveurs Choco2 et AbsCon sur la bibliothèque XCSP 2.1.

1 Introduction

La communauté Programmation Par Contraintes (PPC) dispose de trois principales bibliothèques pour l'évaluation des solveurs : MiniZinc [1], XCSP [2] et la CSPLib [3]. Les bibliothèques MiniZinc et XCSP sont construites autour d'un langage qui permet la représentation de nombreux problèmes sous contraintes. Tandis que MiniZinc permet de séparer modèles et données (tout comme par exemple, OPL), XCSP et FlatZinc permettent de représenter les instances de modèles (dites "à plat").

D'autres communautés proposent des bibliothèques de modèles, par exemple, la MIPLib ou la SATLib. Cependant, la définition d'un langage de modélisation/représentation commun est l'objet d'un plus grand consensus en SAT et MIP qu'en PPC. En effet, en PPC, les contraintes et algorithmes diffèrent significativement d'un solveur à l'autre. De plus, les performances d'un même solveur peuvent varier énormément sur deux modèles logiquement équivalents (et une reformulation efficace, voire automatisée, reste une tâche complexe). Finalement, une modélisation fine adaptée à un solveur peut considérablement améliorer les résultats. Un certain consensus existe malgré tout, notamment sur les principales contraintes globales.

Les bibliothèques de modèles sont couramment utilisées lors de compétitions de solveurs. Les solveurs sont alors évalués le plus souvent sur l'aspect *black-box* et les participants fournissent des efforts importants pour la reformulation automatique du problème ainsi que la configuration automatique du solveur. Par ailleurs, les participants proposent généralement des portfolio lorsque le parallélisme est autorisé.

Ces bibliothèques sont d'un intérêt certain pour l'expérimentation : vaste corpus de modèles et d'instances, comparaison d'algorithmes et reproductibilité des expériences. La bibliothèque la plus aboutie est la MIPLIB qui propose aussi un ensemble de scripts pour la résolution des instances et la compilation des résultats. D'un autre côté, la CSPLib propose des séries de *problèmes* (énoncés et données) pour lesquelles les jeux d'instances et les résultats sont compilés. Cette approche est très répandue en Recherche Opérationnelle : OR-library, TSP, PSP, etc. La ROADEF organise ainsi une compétition de résolution de problèmes où les participants proposent fréquemment des modèles très différents selon les technologies utilisées.

Un problème récurrent est la classification des instances. On peut distinguer deux pratiques courantes (l'une n'excluant pas l'autre) : classification qualitative et classification quantitative. Une classification *qualitative* regroupe les instances selon des caractéristiques communes définies à priori. Une classification *quantitative* regroupe les instances selon leur niveau de difficulté. La seconde pratique est très répandue dans les autres communautés, mais à notre connaissance, les classifications de MiniZinc et XCSP sont qualitatives.

L'intérêt d'une classification croisée, que nous re-

1. http://www.minizinc.org/
2. http://www.cril.fr/~lecoutre/benchmarks.html
3. http://www.csplib.org/

cherchons, est assez évident, à savoir, a) améliorer le processus de développement des solveurs (tests de correction et de performance), b) centrer les compétitions sur les modèles difficiles, c) proposer un corpus de modèles pertinents pour évaluer les nouvelles approches.

2 Bibliothèque XCSP

Le format XCSP 2.1 permet de représenter un réseau de contraintes en XML : instances des cadres CSP (Constraint Satisfaction Problem), Weighted CSP (WCSP) et Quantified CSP (QCSP). Ce format ne permet pas de représenter naturellement les problèmes d'optimisation [4]. Ceux-ci sont généralement transformés en un ou plusieurs CSPs (bornes supérieures ou preuve d'optimalité). Chaque réseau peut impliquer des contraintes en extension, en intension, ou même des contraintes globales (allDifferent, weightedSum, etc). Pour plus d'information, le lecteur pourra se référer au manuel de référence du format XCSP 2.1 [5].

Classification qualitative De nombreux modèles au format XCSP 2.1 ont été rassemblés ces dernières années. On peut les distinguer selon qu'ils soient structurés ou aléatoires. Cependant, cette classification a été affinée en définissant les catégories suivantes : **REAL** (applications réelles), **PATT** (motifs réguliers et avec composantes aléatoires), **ACAD** (académique sans composantes aléatoires), **QRND** (aléatoire et faiblement structuré), **RAND** (complètement aléatoire). On peut approximativement considérer que les modèles sont classés des plus structurés (**REAL**) aux moins structurés (**RAND**).

Choix des instances Nous ne traiterons que les CSPs des catégories **REAL** (1266) et **ACAD** (576), soit 1842 instances parmi presque 10000. De manière subjective, nous avons exclu les autres catégories. La catégorie **PATT** est principalement constitué de COPs transformés en séries d'instances CSP. Les catégories **QRND** et **RAND** ne sont composées que de contraintes en extension, et ces contraintes apparaissent assez fréquemment dans **REAL** et **ACAD**.

3 Classification quantitative

Nous classerons les instances grâce à un portfolio de solveurs Choco2 et AbsCon, chacun effectuant une résolution indépendante (pas de communication). Tous les solveurs utilisent un algorithme standard de retour arrière et l'heuristique de sélection de la valeur minimale.

4. XCSP 3.0 en cours de développement va y remédier.

Solveurs Choco2. `wdeg` : heuristique basée sur le ratio "domaine sur degré pondéré" [1] ; `wdeg-re` : `wdeg` avec une politique de redémarrage à croissance géométrique ; `impact` : heuristique basée sur les impacts [4] ; `impact-re` : `impact` avec redémarrages. Nous n'appliquerons ni reformulation, ni configuration automatique, une approche *blindbox* plutôt que *blackbox*. Aucun *nogood* n'est enregistré lors des redémarrages [2] et certaines parties de l'arbre de recherche peuvent donc être explorées plusieurs fois.

Solveurs AbsCon. En plus des configurations précédentes, `activity` : heuristique basée sur l'activité [3] ; `activity-re` : `activity` avec redémarrages ; `ddeg` : heuristique basée sur le ratio "domaine sur degré dynamique" ; `ddeg-re` : `ddeg` avec redémarrages. Les *nogoods* sont toujours enregistrés lors d'un redémarrage.

Environnement de test On alloue un cœur sur un cluster à chaque solveur pour une durée limitée, **Choco2** processeurs Intel E5-2670 (2.60 GHz – 8 cœurs) et 12 heures ; **AbsCon** processeurs Xéon (2.66 GHz – 2 coeurs) et 1000 secondes.

Nous proposons une classification en fonction des résultats du portfolio de solveurs. Le *virtual worst solver* (VWS) est le temps de résolution du pire solveur, i.e., le temps de résolution séquentielle avec le pire choix. Le *virtual best solver* (VBS) est le temps de résolution du meilleur solveur, i.e., le temps de résolution d'un portfolio parallèle. Les instances **faciles** sont classées avec le **VWS** : **training** (moins de 10 secondes) et **easy** (moins de 1000 secondes). Les instances **difficiles** sont classées avec le **VBS** : **medium** (moins de 1000 secondes) ; **challenging** (pas de timeout) ; **hard** (timeout ou memory exception

Ces catégories forment une partition des instances. La catégorie d'une instance est la première catégorie dont la condition est validée. Ainsi, une instance **medium** est résolue en moins de 1000 secondes par le VBS et en plus de 1000 secondes par le VWS (si ce n'était pas le cas pour cette seconde condition, elle serait **easy**). Bien sûr, une telle classification doit être étendue pour compiler des résultats venant de protocoles expérimentaux plus variés (solveurs, matériels, etc). Cependant, si le protocole est discutable pour identifier les modèles difficiles, il permet certainement d'identifier des modèles faciles.

4 Satisfaction de contraintes

La figure 1 permet de visualiser les performances globales du portfolio et celles des portfolio restreints à Choco2 et AbsCon. Pour chaque portfolio, le nombre d'instances résolues par le VBS (resp. VWS) est tracé

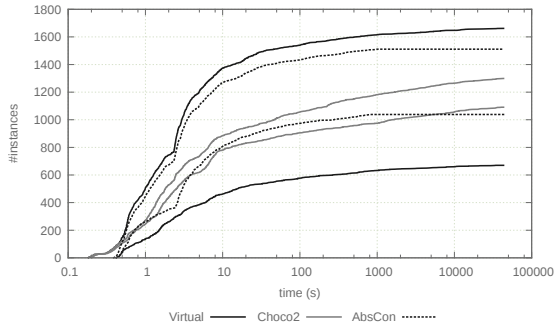


FIGURE 1 – Performances du portfolio.

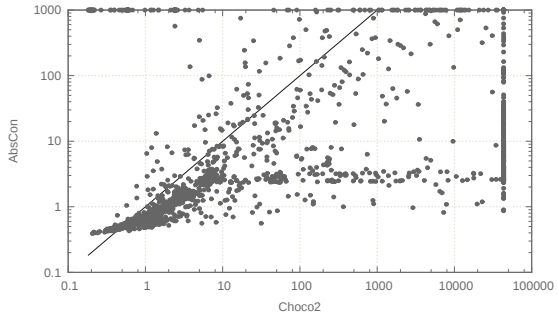


FIGURE 2 – Choco2 versus AbsCon

comme une fonction du temps de résolution. On peut observer que *75% des instances sont résolues en moins de 10 secondes par le VBS et 25% par le VWS*. Par ailleurs, plus le portfolio a de solveurs, plus l'écart entre le VBS et le VWS est grand. Le portfolio AbsCon résout plus d'instances que le portfolio Choco2, mais ces deux solveurs ne résolvent pas les mêmes instances (c.f. l'écart avec le VWS). Ainsi, l'intérêt du portfolio est démontré pour mesurer la difficulté des modèles.

La figure 2 décrit plus en détail les résultats des portfolio Choco2 et AbsCon. Chaque instance est représentée par un point dont la coordonnée x est le temps de résolution du portfolio Choco2 et la coordonnée y est le temps de résolution du portfolio AbsCon. Les échelles des deux axes sont logarithmiques. Tous les points situés sous (resp. au dessus de) la diagonale ($x = y$) représentent des instances résolues plus rapidement avec AbsCon (resp. Choco2). Les points formant une ligne verticale à droite (resp. horizontale en haut) sont les instances résolues seulement avec AbsCon (resp. Choco2). La figure confirme que Choco2 et AbsCon ont des comportements différents.

La figure 3 montre l'influence positive des redémarrages. Chaque triplet (instance, solveur, branchement) est représenté par un point dont la coordonnée x est le temps de résolution sans redémarrage et la coordonnée y est le temps de résolution avec redémarrage. Une

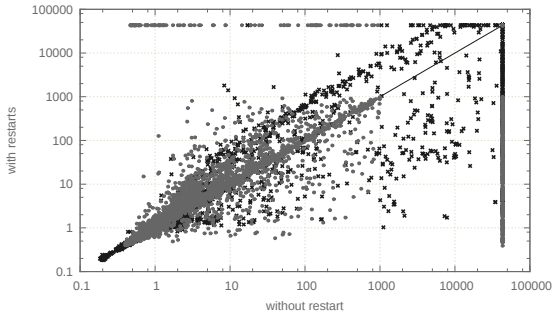


FIGURE 3 – Importance des redémarrages.

croix (resp. un cercle) indique que le solveur utilisé est Choco2 (resp. AbsCon). Les redémarrages améliorent globalement la résolution en permettant de résoudre un certain nombre d'instances supplémentaires dans le temps imparti. De plus, la dégradation des temps de résolution, dûe aux redémarrages, reste raisonnable, alors que les gains sont souvent importants. Dans le pire cas, les redémarrages détériorent plus la résolution avec AbsCon qu'avec Choco2 (ligne horizontale en haut), malgré les nogoods.

Le tableau 1 donne le nombre d'instances par classe. Plus de 600 instances sont quasiment triviales et seulement 226 instances sont réellement difficiles, dont 46 sont résolues par le portfolio.

5 Énumération de solutions

Notre but est maintenant de sélectionner des instances dont il est difficile d'énumérer les solutions (mais pour lesquelles c'est encore possible). Nous utiliserons uniquement Choco2 sans redémarrage sur 569 instances faciles, i.e., dont la première solution est trouvée en moins de deux minutes par le VWS Choco2. Tout d'abord, la tâche s'est révélée impossible pour 387 instances qui avait souvent des milliards de solutions ($> 2^{32}$). Au contraire, la tâche s'est avérée facile pour 153 instances résolues à nouveau en moins de deux minutes. Au final, seules subsistent 23 instances, dont certaines avec plus de 10^{12} solutions. Finalement, le tableau 2 indique les 15 instances difficiles pour le

	train.	easy	med.	chall.	hard	**total**
ACAD	144	48	233	23	128	576
REAL	318	122	751	23	52	1266
UNSAT	135	85	471	24	–	715
SAT	327	85	513	22	–	947
total	462	170	984	46	180	1842

TABLE 1 – Classification quantitative

Instance	#sol.	VBS
crossword-m1c-words-vg5-7_ext	14	562.3
crossword-m1-words-05-02	1025730	586.4
crossword-m1c-words-vg5-6_ext	2308	596.4
crossword-m1c-words-vg7-7_ext	6	908.3
crossword-m1-words-05-06	3298018	1094.4
crossword-m1c-words-vg6-6_ext	5845	1125.7
squares-9-9	9456120	1377.6
costasArray-13	12828	1447.7
series-14	9912	3386.1
queenAttacking-6	2678	3781.7
queens-15	2279184	3888.1
ortholatin-5	6220800	4881.2
costasArray-14	17252	7351.8
langford-3-17	26880	11715.5
costasArray-15	19612	37774.2

TABLE 2 – 15 instances pour l'énumération.

VBS avec moins de 10 millions de solutions (temps pour le VBS donné en secondes). On notera que l'influence de l'heuristique de sélection de variables est réduite. En effet, la moyenne géométrique du ratio des temps de résolution entre `impact` et `wdeg` est 0.996 avec un écart type de 0.105.

6 Décompte des points

Nous profitons des données fournies par cette campagne expérimentale intensive pour donner à titre indicatif un classement des solveurs les plus robustes par rapport à la bibliothèque XCSP. Pour classer les solveurs en fonction de leur performance, nous utilisons un principe de décompte des points basé sur la méthode Borda (système de vote pondéré). Chaque instance est un électeur qui classe les différents solveurs. Un solveur marque un nombre de points dépendant du nombre de solveurs qu'il bat. Un solveur s marque des points sur une instance i en comparant ses performances avec celles de chaque autre solveur s' : a) 1 point si s donne une meilleure réponse que s', b) si les réponses sont équivalentes, les points sont partagés, c) 0 point si s' donne une meilleure réponse. Quand les réponses de s et s' sont équivalentes, s marque $f(t,t') = t' \div (t+t')$ point ($f(0,0) = 0$). Nous proposons une nouvelle fonction $g(t,t') = g(t) + (1 - g(t) - g(t')) \times f(t,t')$ dans laquelle une partie des points est accordée à un solveur par contrat $g(t) = 1 \div 2 \times (\log_a(t+1)+1)$ ($a = 10$). Les points restants sont partagés entre les deux solveurs grâce à la fonction f.

Le tableau 3 donne les classements selon la fonction utilisée. Les deux premières colonnes (s) sont le nombre d'instances résolues dans la limite de temps par chaque solveur, une mesure de performance essentielle. Les changements pertinents induits par g sont indiqués en gras. Ainsi, le biais de la fonction f en faveur des solveurs rapides sur les instances faciles est réduit par la fonction g. Au sens des moindre carrés sur les classements, g est plus proche de s que f ($\|g-s\|_2 = 12$ et $\|f-s\|_2 = 30$. A noter que le suffixe `-re` signale une version avec redémarrages.

	s		Class. s		Class. f		Class. g	
	*	*-re	*	*-re	*	*-re	*	*-re
impact	1119	1196	14	12	14	12	14	12
wdeg	1156	1259	13	10	13	11	13	11
Choco2	**1299**		7		**10**		**8**	
activity	1337	1405	6	3	5	4	5	4
ddeg	**1275**	**1246**	9	11	**6**	**9**	**7**	**10**
impact	**1288**	**1372**	8	5	**8**	**7**	**9**	**6**
wdeg	1376	1465	4	2	3	2	3	2
AbsCon	1511		1		1		1	

TABLE 3 – Classements des solveurs.

7 Conclusion

Dans cet article, nous nous sommes intéressés à la question délicate de la classification quantitative des modèles (i.e., selon leur difficulté). Sur la base d'une approche portfolio, nous avons étudié les instances de deux catégories importantes de la bibliothèque XCSP. Ce travail sera utilisé et approfondi lors de la mise en place du nouveau site web, développé conjointement au nouveau format étendu de représentation XCSP 3.0, programmée courant 2014.

Remerciements

Ces travaux bénéficient du soutien du CNRS et d'OSEO dans le cadre du projet ISI "Pajero" et d'un accès aux moyens de calculs et de visualisation du Centre de Calcul Interactif hébergé à l'Université Nice Sophia Antipolis.

Références

[1] F. Boussemart, F. Hemery, C. Lecoutre, and L. Sais. Boosting systematic search by weighting constraints. *Actes de ECAI'04*, pages 146–150, 2004.

[2] C. Lecoutre, L. Sais, S. Tabary, and V. Vidal. Nogood recording from restarts. *Actes de IJCAI'07*, pages 131–136, 2007.

[3] L. Michel and P. Van Hentenryck. Activity-based search for black-box constraint programming solvers. *Actes de CPAIOR'12*, pages 228–243, 2012.

[4] P. Refalo. Impact-based search strategies for constraint programming. *Actes de CP'04*, pages 557–571, 2004.

[5] O. Roussel and C. Lecoutre. XML representation of constraint networks : Format XCSP 2.1. Technical Report arXiv :0902.2362, CoRR, 2009.